Karlsruher Institut für Technologie

Advanced Cryptographic Techniques for Protecting Log Data

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Gunnar Richard Hartung

Tag der mündlichen Prüfung:	28. Oktober 2019
Erster Referent:	Prof. Dr. Jörn Müller-Quade
Zweiter Referent:	Prof. Dr. Tibor Jager

Abstract

This thesis examines cryptographic techniques providing security for computer log files. It focuses on ensuring authenticity and integrity, i.e. the properties of having been created by a specific entity and being unmodified. Confidentiality, the property of being unknown to unauthorized entities, will be considered, too, but with less emphasis.

Computer log files are recordings of actions performed and events encountered in computer systems. While the complexity of computer systems is steadily growing, it is increasingly difficult to predict how a given system will behave under certain conditions, or to retrospectively reconstruct and explain which events and conditions led to a specific behavior. Computer log files help to mitigate the problem of retracing a system's behavior retrospectively by providing a (usually chronological) view of events and actions encountered in a system.

Authenticity and integrity of computer log files are widely recognized security requirements, see e.g. [Lat85, p. 10], [KS06, Section 2.3.2], [GR95, Section 18.3.1], [NDP17, Section 9.3], [CC12, Section 8.6]. Two commonly cited ways to ensure integrity of log files are to store log data on so-called write-once-read-many-times (WORM) drives and to immediately print log records on a continuous-feed printer. This guarantees that log data cannot be retroactively modified by an attacker without physical access to the storage medium.

However, such special-purpose hardware may not always be a viable option for the application at hand, for example because it may be too costly. In such cases, the integrity and authenticity of log records must be ensured via other means, e.g. with cryptographic techniques. Although these techniques cannot *prevent* the modification of log data, they can offer strong guarantees that modifications will be *detectable*, while being implementable in software. Furthermore, cryptography can be used to achieve *public* verifiability of log files, which may be needed in applications that have strong transparency requirements. Cryptographic techniques can even be used *in addition to* hardware solutions, providing protection against attackers who *do* have physical access to the logging hardware, such as insiders.

Cryptographic schemes for protecting stored log data need to be resilient against attackers who obtain control over the computer storing the log data. If this computer operates in a standalone fashion, it is an absolute requirement for the cryptographic schemes to offer security *even in the event of a key compromise.* As this is impossible with standard cryptographic tools, cryptographic solutions for protecting log data typically make use of *forward-secure* schemes, guaranteeing that changes to log data recorded in the past can be detected. Such schemes use a *sequence* of authentication keys instead of a single one, where previous keys cannot be computed efficiently from latter ones.

This thesis considers the following requirements for, and desirable features of, cryptographic logging schemes: 1) security, i.e. the ability to reliably detect violations of integrity and authenticity, including detection of log truncations, 2) efficiency regarding both computational and storage overhead, 3) robustness, i.e. the ability to verify unmodified log entries even if others have been illicitly changed, and 4) verifiability of excerpts, including checking an excerpt for omissions.

The goals of this thesis are to devise new techniques for the construction of cryptographic schemes that provide security for computer log files, to give concrete constructions of such schemes, to develop new models that can accurately capture the security guarantees offered by the new schemes, as well as to examine the security of previously published schemes.

This thesis demands that cryptographic schemes for securely storing log data must be able to detect if log entries have been deleted from a log file. A special case of deletion is log *truncation*, where a continuous subsequence of log records from the end of the log file is deleted. Obtaining *truncation resistance*, i.e. the ability to detect truncations, is one of the major difficulties when designing cryptographic logging schemes. This thesis alleviates this problem by introducing a novel technique to detect log truncations without the help of third parties or designated logging hardware. Moreover, this work presents new formal security notions capturing truncation resistance. The technique mentioned above is applied to obtain cryptographic logging schemes which can be shown to satisfy these notions under mild assumptions, making them the first schemes with formally proven truncation security.

Furthermore, this thesis develops a cryptographic scheme for the protection of log files which can support the creation of *excerpts*. For this thesis, an excerpt is a (not necessarily contiguous) subsequence of records from a log file. Excerpts created with the scheme presented in this thesis can be publicly checked for *integrity and authenticity* (as explained above) as well as for *completeness*, i.e. the property that no relevant log entry has been omitted from the excerpt. Excerpts provide a natural way to preserve the confidentiality of information that is contained in a log file, but not of interest for a specific public analysis of the log file, enabling the owner of the log file to meet confidentiality and transparency requirements at the same time. The scheme demonstrates and exemplifies the technique for obtaining truncation security mentioned above.

Since cryptographic techniques to safeguard log files usually require authenticating log entries individually, some researchers [MT08; MT09; YP09] have proposed using aggregatable signatures [Bon+03] in order to reduce the overhead in storage space incurred by using such a cryptographic scheme. *Aggregation* of signatures refers to some "combination" of any number of signatures (for distinct or equal messages, by distinct or identical signers) into an "aggregate" signature. The size of the aggregate signature should be less than the total of the sizes of the orginal signatures, ideally the size of *one* of the original signatures. Using aggregation of signatures in applications that require storing or transmitting a large number of signatures (such as the storage of log records) can lead to significant reductions in the use of storage space and bandwidth.

However, aggregating the signatures for all log records into a single signature will cause some *fragility*: The modification of a single log entry will render the aggregate signature invalid, preventing the cryptographic verification of any part of the log file. However, being able to distinguish manipulated log entries from non-manipulated ones may be of importance for after-the-fact investigations. This thesis addresses this issue by presenting a new technique providing a trade-off between storage overhead and *robustness*, i.e. the ability to tolerate some modifications to the log file while preserving the cryptographic verifiability of unmodified log entries. This robustness is achieved by the use of a special kind of aggregate signatures (called *fault-tolerant* aggregate signatures), which contain some redundancy. The construction makes use of combinatorial methods guaranteeing that if the number of errors is below a certain threshold, then there will be enough redundancy to identify and verify the non-modified log entries.

Finally, this thesis presents a total of four attacks on three different schemes intended for securely storing log files presented in the literature [YPR12b; Ma08]. The attacks allow for virtually arbitrary log file forgeries or even recovery of the secret key used for authenticating the log file, which could then be used for mostly arbitrary log file forgeries, too. All of these attacks exploit weaknesses of the specific schemes. Three of the attacks presented here contradict the security properties of the schemes claimed and supposedly proven by the respective authors. This thesis briefly discusses these proofs and points out their flaws. The fourth attack presented here is outside of the security model considered by the scheme's authors, but nonetheless presents a realistic threat.

In summary, this thesis advances the scientific state-of-the-art with regard to providing security for computer log files in a number of ways: by introducing a new technique for obtaining security against log truncations, by providing the first scheme where excerpts from log files can be verified for completeness, by describing the first scheme that can achieve some notion of robustness while being able to aggregate log record signatures, and by analyzing the security of previously proposed schemes.

Contents

1. Introduction

Computer log files are recordings of actions performed and events encountered in computer systems. As the complexity of computer systems is steadily accumulating, it is increasingly difficult to predict how a given system will behave under certain conditions, or to retrospectively reconstruct and explain which events led to a specific behavior. The problem is even worse for systems based on machine learning techniques, where even experts struggle to understand and explain the precise rules and/or parameters "learned" by a system from some training data.

At the same time, dependency on complex computer systems is increasing, and more and more valuable and invaluable assets are entrusted to computers and the algorithms they are running. For example, the United States' Commodity Futures Trading Commission (CFTC, a US government agency supervising and regulating a specific branch of finance) reported that in the time period from November 2014 until October 2016, 77.7% of the total market volume for so-called "futures" trading US dollars for Japanese Yen could be attributed to trades where the offer was placed and accepted by automated trading systems [HR17]. An additional 18.8% of the market volume was due to contracts where either the placement or the acceptance was the result of an automated decision. The report noted [HR17]:

> The level of automated trading has increased, over the past few years, across all of the major product groups traded on the [Chicago Mercantile Exchange].

Moreover, human lives implicitly depend on the correct functioning of computer systems on a daily basis. Pacemakers, fully automated trains or trams, cars' onboard control systems, and aerial traffic monitoring systems are just a few examples of computer systems whose malfunction could endanger human lives. The enormous degree of pervasion of today's society by computers of all kinds has led to a strong dependence on these computers, which gives rise to the need for accountability.

However, the enormous complexity of today's systems is a serious hindrance for establishing accountability. Computer log files alleviate the burden of complexity during reviews of past system behavior by providing a (usually chronological) record of events and actions (such as business trancations, errors or security violations) which happened in a computer system. They are an indispensable source of information for after-the-fact digital forensics, system maintenance, as well as intrusion detection. For all of these objectives, having reliable information is imperative.

Therefore, authenticity and integrity of computer log files are widely recognized security requirements. *Authenticity* refers to the property of originating from a specific

entity, e.g. a person, a computer or an organisation. *Integrity* refers to the property of being unmodified with respect to some reference version.

For example, the "Orange Book" [Lat85], published by the United States' Department of Defense in 1985, a collection of security requirements for products used to process classified or otherwise sensitive information, stated [Lat85, p. 10]:

> A trusted system must be able to record the occurrences of security-relevant events in an audit log. [...] Audit data must be protected from modification and unauthorized destruction to permit detection and after-the-fact investigations of security violations.

Similarly, the United States' National Institute of Standards and Technology (NIST) claimed, in the handbook "An Introduction to Computer Security" published in 1995 [GR95, Section 18.3.1]:

> It is particularly important to ensure the *integrity* of audit trail data against modification. One way to do this is to use digital signatures. [...] Another way is to use write-once devices. The audit trail files needs [sic!] to be protected since, for example, intruders may try to "cover their tracks" by modifying audit trail records.

While the handbook above has recently been withdrawn and superseded, the superseding version [NDP17, Section 9.3] suggests:

> Cryptography may play a useful role in audit trails, which are used to help support electronic signatures. Audit records may implement electronic signatures for integrity, and cryptography may be needed to protect audit records stored on systems from disclosure or modification.

In another publication, dealing specifically with logs in the context of computer security, NIST states [KS06, Section 2.3.2]:

> Because logs contain records of system and network security, they need to be protected from breaches of their confidentiality and integrity. [...] Logs that are secured improperly in storage or in transit might also be susceptible to intentional and unintentional alteration and destruction. This could cause a variety of impacts, including allowing malicious activities to go unnoticed and manipulating evidence to conceal the identity of a malicious party. For example, many rootkits are specifically designed to alter logs to remove any evidence of the rootkits' installation or execution.

Furthermore, the "IT-Grundschutz"—a set of standards on how to manage IT security in companies or government agencies developed by the German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik, BSI)—explicitly names "Loss of Confidentiality and Integrity of Log Data" (author's translation) as a threat to be countered [Bun19, Module OPS.1.1.5, Section 2.5]. Additionally, the standard contains the following suggestion for IT systems with a high need of protection: [Bun19, OPS.1.1.5.A12, author's translation]

> [...] Furthermore, all stored logs SHOULD be signed digitally.

Given these recommendations, it is not surprising that the so-called "Common Criteria" specify requirements for the secure storage of audit data. [CC12, Section 8.6].

Approaches to Securely Storing Log Data. Two commonly cited methods of ensuring integrity of log files are to store log data on so-called write-once-read-many-times (WORM) drives (as noted by [GR95]) and to immediately print log records on a continuous feed printer. These methods guarantee that log data cannot be retroactively modified by an attacker without physical access to the storage media.

Another mechanism, which is frequently used in practice, is to send log records to a dedicated logging server over a network. While this approach may be an improvement in terms of security if the logging server is well secured and hardened against attacks, the problem of securely *storing* log files is ultimately just shifted to a different part of the system. Moreover, this approach requires that the *transmission* of log data is secure and reliable. While cryptography has developed a rich set of tools for *secure* transmission of data over untrusted networks (e.g. TLS [RFC8446]), most modern computer networks do not offer strict guarantees of reliability.

Yet another approach to ensuring the authenticity and integrity of log files is to use cryptography, as proposed by [GR95; NDP17]. While cryptographic techniques cannot *prevent* the modification of log data, they can offer strong guarantees that modifications will be *detectable*.

This thesis addresses the problem of securely storing log files, taking the latter approach. It focuses on the case where neither specialized hardware nor other servers are available (i.e. the logging system must operate in a completely standalone fashion after deployment). This setting is referred to as the *standalone model*. Our reasons for studying the security of log files from a cryptographic perspective and in the standalone model are as follows:

- Cryptographic solutions can achieve *public verifiability* of log files, i.e. the property that (at least in principle) everyone can check the log file for her-/himself. This feature may be needed in applications that have strong transparency requirements (such as electronic voting), or when log data is introduced into a lawsuit as evidence.

- Cryptographic techniques can be implemented purely in software, without requiring additional hardware (such as WORM drives, continuous feed printers or dedicated log storage servers).

- Cryptography can even achieve security against *insider threats* (e.g. rogue employees) who have access to the log storage media.

- If so desired, cryptographic techniques for securely storing log data can be used *in addition to* hardware-based approaches such as WORM drives and profit from the capabilities offered by the hardware solution.

- Since cryptographic solutions can be implemented in software, their implementation is more amenable to analysis and inspection than hardware solutions.
- The security of abstract cryptographic solutions can be formally proven (with regard to mathematical definitions of security), under certain computational assumptions.
- The science of cryptography is often concerned with determining the weakest assumption(s) necessary to achieve certain security goals. By restricting our attention to the standalone model and refraining from assuming the presence of designated hardware modules we (indirectly) address this question for the case of protecting stored log data.

Desiderata for Cryptographic Logging Schemes. This work considers the following requirements, features and properties for assessing the quality of cryptographic logging schemes:

- First and foremost, a cryptographic logging scheme must be *secure*. In particular, it is required that the logging scheme protects the integrity and authenticity of stored log data. This includes that attackers must not be able to modify existing log records, inject new, forged ones, delete existing log entries, or re-order the existing log records without being detected.

  In the standalone setting considered in this thesis, where the data logger cannot rely on designated hardware or external servers, the cryptographic keys used to authenticate the log data must be kept on the same computer that also stores the log data. Hence, when an attacker compromises this system in order to manipulate stored log data, the attacker can also obtain the authentication key.[1] Thus, the cryptographic schemes employed must continue to offer security *even in the event of a key compromise.* Cryptographic schemes having this property are called *forward-secure.*

- This work considers the *efficiency* of cryptographic logging schemes, i.e., the overhead introduced by the secure storage of log files as opposed to storing them without protection. Efficiency itself can be measured in a number of ways, such as the the required running time for generating keys, authenticating log records or verifying a log file, as well as the storage requirements for cryptographic keys and signatures.

  Naturally, when comparing the efficiency of logging schemes, one scheme's efficiency may exceed another's in some of these aspects, while falling short in other aspects. In these cases, it is difficult to say that one scheme is more efficient than

[1] We conservatively assume that any compromise of the machine storing the log records is complete, i.e. the attacker gains unrestricted access to the machine. This is in accordance with the overall cryptographic approach taken in this thesis: We strive to give mathematical guarantees of security, and to refrain making assumptions on the attacker's ability to corrupt a given system.

another one. The logging schemes constructed in this thesis strike a reasonable balance between different aspects of efficiency.

It may at times be desirable to verify individual log entries with as little overhead as possible. In particular, one may want to check individual log entries without needing to verify the entire log file. A scheme's ability to perform such a *selective verification* is considered as another (slightly "exotic") aspect of efficiency.

- Cryptographic logging schemes should have *robustness*, i.e. the ability to verify the integrity and authenticity of parts of a log file, even if other parts have been illegitimately manipulated. Otherwise, once a compromise is detected, it may be impossible to tell which log records are true and which ones have been manipulated, severely obstructing the after-the-fact forensic analysis of the incident.

  While it may be possible to consult backup copies of the stored log data, such backup copies need not contain the authentic, "real" log data, neither, if the tampering has occured a long time before its detection.

- This thesis considers the feature of providing *excerpts* of log files, i.e. subsequences of log records from a log file. It should be possible to verify these excerpts for authenticity and integrity (log data contained in the excerpts should be "correct") as well as *completeness*. Completeness refers to the property of containing all "relevant" log entries, as opposed to missing some (potentially critical) log records.

Goals of this Thesis. Following the cryptographic approach to securely storing log data, the goals of this thesis are:

- to devise new techniques for the construction of cryptographic schemes that provide security for computer log files,
- to provide concrete constructions of such schemes following the desiderata laid out above,
- to develop new models that can accurately capture the security guarantees offered by the new schemes,
- to conduct proofs of security for these schemes in the respective models, relating the difficulty of retroactively forging log entries to the hardness of well-defined, well-studied and assumedly intractable mathematical problems, as well as
- to examine the security of previously published schemes.

Practical Considerations. In the field of IT security, it is a common approach to monitor log data for intrusion detection. If log data is cryptographically authenticated, the monitoring should involve regularly verifying the integrity and authenticity of the log data. Doing so will detect modifications that attackers may have performed in order to hide their activities.

Once modifications have been detected, it is usually clear that a security violation has occured, and appropriate actions should be taken to react to this violation. How to respond to such a situation is outside the scope of this thesis, but the reaction will typically contain an ex post examination of the incident. During this examination, one may (and should) consider existing backup copies of the log files. This might help to recover from modifications of log data recorded before the creation of the latest backup copy.

1.1. Foundations

This section briefly and informally recaps some cryptographic concepts and techniques which are prevalent in the study of cryptographic schemes for protecting stored log data.

This section is strongly based [Har16b, Section 1]. Significant parts of the text in this section have been taken from that work without or with only minor modifications and without specific designation. In addition, this section reproduces text from [Har17, Section 1] and [Har+17b, Section 2], again without designation.

Digital Signatures and MACs. The standard cryptographic approach to ensure authenticity and integrity of some information is to authenticate the data at hand using digital signature schemes (a concept initially conceived by Diffie and Hellman [DH76]) or message authentication codes (MACs). Although the techniques presented in this work can be applied to MACs as well, this work focuses on the case of digital signatures, since they allow for public verification of the authenticity and integrity of signed data. MACs, in contrast, are not publicly verifiable in general.

Digital signatures are commonly modeled as triples of algorithms: key generation, message signing, and verification.[2] Before any data can be authenticated, the originator (or *sender*) of that data runs the key generation algorithm to obtain an authentication key and a verification key. The authentication key must be kept secret (hence, it is called the *secret key*), whereas the verification key can be published (*public key*). In order to authenticate some target datum (the *message*), the sender runs the signing algorithm (using the secret key) in order to derive an additional datum (the *signature*), which serves as a cryptographic proof that the message at hand indeed originates from the sender. The signature is then transmitted to the intended recipient alongside the message. Upon receiving a message and an ostensible signature the recipient may execute the verification algorithm to determine if the received datum indeed originates from the sender. Running the verification algorithm requires the sender's public verification key, the received datum and the signature generated by the sender.

[2] Kerckhoffs' principle ([Ker83], see also [KL07, pp. 6–8]) mandates that these algorithms are publicly known. While Kerckhoffs' principle predates the invention of digital signatures by almost a century and was originally only meant to apply to encryption systems, it applies analogously to other types of cryptographic schemes, such as digital signatures. Kerckhoffs' principle has developed to be one of the fundamental and most universally applied principles of modern cryptography.

A signature accepted by the verification procedure is said to be *valid* for the respective message (under the respective public key).

MAC schemes are very similar to digital signature schemes, but differ in the fact that they use a *single* key for both authenticating and verifying messages. Therefore, this key must be kept secret and hence the verification algorithm cannot be run publicly.

In order to provide adequate assurance of data authenticity, it must be very hard for an attacker to forge a signature confirming the authenticity of a message which does not actually originate from the sender. Constructing digital signature schemes in a way that makes forgeries very hard is a frequent subject of research in cryptography.

On an intuitive level, the de facto standard definition of security [GMR84; GMR88] for digital signatures requires that it should be "practically impossible" to create a valid signature for some message under a given public key without knowing the corresponding secret key, even if an attacker knows valid signatures for different messages selected by the attacker her-/himself. This, together with the implicit assumption that the secret key is only known to the sender, implies that if a message has a valid signature, then only the sender could have created the signature, and thus the message must indeed originate from the sender.

Forward Security. In the context of this work, dealing with protecting log files in a stand-alone setting, it is necessary to consider the possibility that the data logger might get corrupted by the attacker at some point in time. (If the data logger was considered to be incorruptible, the attacker would not be able to manipulate the logged data in the first place.) Once an attacker has taken control over a system, (s)he may access all cryptographic keys stored within that system, including the secret keys used to sign log data.

Using the keys and the publicly known signature algorithm, an attacker can compute signatures on her/his own. Thus, an attacker might modify log data arbitrarily, and recompute a valid signature for the forged log data afterwards. Since the signature will be valid, such a modification will not be detectable by executing the verification algorithm.

Therefore, cryptographic solutions for protecting stored log data must be resilient to attackers who gain full control of the log server which holds the secret key. Rephrasing this requirement, a cryptographic logging scheme must remain secure *even if the attacker obtains the secret key* at some point in time, and must continue to enable the discovery of illicit, retroactive modifications of log records.

As this is impossible with standard authentication schemes, researchers have devised schemes (e.g. [BY97; BM99; BY03; AR00; IR01; MMM02; Kra00; Boy+06; Son01; ZWW03; AMN01; HWI03]) which guarantee "forward integrity" [BY97].[3] Such schemes use *a series* of secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_T$ (instead of a single constant secret key) for authentication and integrity protection, where each key sk_i is used for some time period

[3] The term "forward integrity" as introduced by [BY97] referred to MAC schemes only. However, the principle applies analogously to digital signature schemes.

(called the *i-th epoch*), until it is eventually erased and replaced by its successor.[4] The key sk_{i+1} may either be computed from sk_i in a completely deterministic fashion, or may be chosen at random, depending on the scheme at hand. The verification algorithm is adapted to accept an index $t \in \{1, \ldots, T\}$ as an additional input and is expected to check whether the message was signed using key sk_t. The verification should fail if the message has not been authenticated at all or has been authenticated under a different key sk_i with $i \neq t$.

Informally speaking, a scheme has forward integrity if obtaining one of these secret keys sk_B does not help in forging a signature with respect to any previous key sk_i with $i < B$. This implies that given sk_B, it must be hard to compute sk_i, since otherwise an attacker could use sk_i and the normal signing procedure to create a forged signature for epoch i. Digital signature schemes as well as MACs that have forward integrity are also called *forward-secure*.

Secure Storage of Log Files. Given a forward-secure signature scheme, one might build a secure log file (or *secure audit log*) as follows [BY97]: When a new log file is created, the scheme generates a key pair $(\mathsf{sk}_1, \mathsf{pk})$. The public key is copied and either published or distributed to a set of verifiers (e.g. auditors). When a new log entry m_1 is added to the log file, the log record is signed with key sk_1, and the resulting signature σ_1 is stored along with the log file. When another log entry m_2 is added, it is signed, too, and the new signature σ_2 is stored together with σ_1 and the messages m_1, m_2. This process continues analogously for newly arriving log messages. At some point in time (for example after a certain amount of time has passed or a certain number of log entries have been signed), the signer updates the secret key sk_1 to sk_2, securely erases sk_1 and continues signing log entries with sk_2 instead of sk_1. At a later point in time, the signer updates sk_2 to sk_3, deletes sk_2 and continues to work with sk_3, and so on.[5] When the log file needs to be verified later, everyone who is in possession of pk (or can securely retrieve a copy of it) can run the verification algorithm to see if the log file has been tampered with.

When an attacker $\mathcal{A}$ takes control over the system during epoch B (and hence obtains the secret key sk_B), the forward security of the applied digital signature scheme guarantees that $\mathcal{A}$ cannot forge valid signatures for log entries created in previous epochs, and thus cannot modify these log records without being detected.

Note that $\mathcal{A}$ can trivially forge signatures for the current epoch B and all future epochs $i > B$ by using the regular signing and updating procedures. Thus, this model cannot offer any guarantees of integrity or authenticity for log data recorded after the attacker has successfully breached the system.

The setting described above defines the security model used throughout this thesis. This model is illustrated by Figure 1.1.

[4] Erasure of secret keys must be complete and irrecoverable to guarantee security, i.e., the secret keys must actually be overwritten or destroyed, instead of just removing (file) pointers or links to the secret key.

[5] Again, the secret keys must be deleted in a way that guarantees that they cannot be recovered.

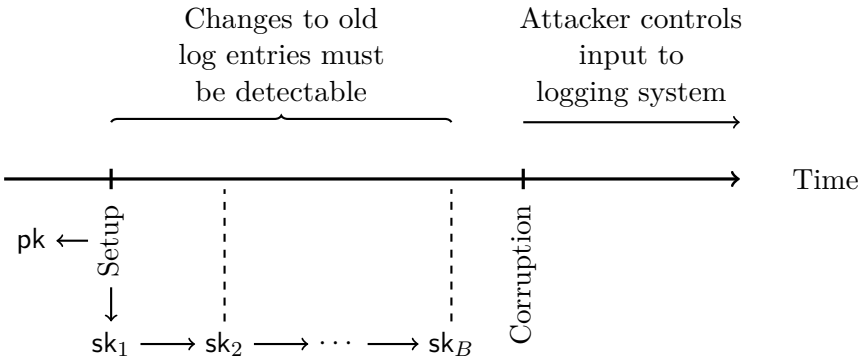

Figure 1.1.: An illustration of the security model considered in this thesis. When the logging system is set up, one generates a pair of keys for a forward-secure signature scheme. The public key is copied and stored in a secure location. Log records are signed using sk_1 at first, until sk_1 is eventually retired, replaced by sk_2, and securely erased. New log records are signed with sk_2, until sk_2 is replaced by sk_3 at the start of the third epoch, and so on. If the attacker $\mathcal{A}$ eventually breaks in during an epoch B and obtains the secret key sk_B, then any changes to log records saved before epoch B should be detectable. After the break-in, the attacker is complete control of the *input* to the logging system, and can fully determine what data is recorded.

However, note that *even if* the log data was stored on a WORM drive or immediately transmitted to another (more secure) server, the attacker would still be in control of the *input* to the logging system, and thus of the logged data. Thus, the lack of cryptographic security guarantees for data logged after system compromise does not lead to any weaknesses that would not exist otherwise.

The scheme sketched above is highly simplified, and can only detect attackers modifying existing log records or adding new ones to epochs before the corruption. However, it is *not* sufficient to protect the integrity of the log file as a whole: For example, an attacker might reorder existing log entries, feigning a sequence of events that is different from what really happened (*reordering attack*). Moreover, an attacker might delete one or more individual log records along with their signatures. Deleting one or more messages from the end of the log file (so that the deleted messages form a suffix of the sequence of real log records) is called a *truncation attack*. Therefore, actual proposals in the literature usually employ a combination of additional measures, which will be detailed further below.

Signature Aggregation. Since log files accumulate log entries over potentially long periods of time, the number of signatures which must be stored to verify the log messages will grow accordingly. In order to improve storage efficiency, it is therefore desirable to be able to "compress" the signatures.

Aggregate signature schemes [Bon+03] allow the signer to merge independently created signatures σ_i on different messages m_i (possibly even originating from different signers, having public keys pk_i) into just one signature σ, which may be as small as a signature for a single message.

For example, in the BGLS aggregate signature scheme [Bon+03], each signature σ_i is a single element of a cyclic group $\mathbb{G}$ with a bilinear map. Signatures are aggregated by simply multiplying them (in $\mathbb{G}$). The resulting signature $\sigma = \prod \sigma_i$ is called the *aggregate* of all σ_i. Using aggregate signatures for secure logging does not only improve the logging system's storage efficiency, but may also help preventing truncation attacks [MT08].

In the BGLS aggregate signature scheme, aggregation is very flexible and can be done in a completely ad-hoc fashion: The commutativity and associativity of the multiplication in the group guarantee that signatures can be aggregated in any order, and aggregated signatures can be aggregated further.

Sequential aggregate signatures [Lys+04] do not support this fully flexible aggregation: Messages are added to an aggregate one-by-one, each message by its signer. Signing and aggregation may be a single, inseparable process (i.e., once created, signatures cannot be combined further in general). Sequential aggregate signatures are not as flexible as general aggregate signatures, but are still useful in a wide range of applications, such as certificate chains, secure routing, version control systems, and securing log files [MT09]. Ma and Tsudik [MT07b] introduced the abbreviation *FssAgg* for forward-secure sequential aggregate signatures. This thesis will additionally use the shorthand *SAS* to refer to sequentially aggregate signatures.

1.2. State of the Art and Related Work

We now briefly review prior work on cryptographic techniques for secure logging. The review presented here is strongly based on [Har16b, Section 1]. Significant parts of the review given here are taken from this publication, without or with only minor modifications, and without further designation.

1.2.1. Common Techniques for Secure Logging

Before moving to a more detailed review of previous work, the author would like to highlight some common themes and techniques which can be found in previous publications:

- Even when not ensuring log file authenticity and integrity, it is common practice to store log entries together with a timestamp. From a security point of view, this enables the detection of reordering attacks, provided the time stamp is signed together with the log record and the timestamp resolution is sufficiently high. [BY97; MT08]

- Many schemes count the number of log entries and add the counter values (*sequence numbers*) to the log record before signing it. This helps to determine

the order of log entries (which reflect real events in the system) if the log entries do not contain timestamps themselves (or the timestamps have too coarse resolution). In addition, this helps to detect deletion of log entries, except for truncation attacks. [BY97; MT08; YP09; YPR12b]

- Some authors (e.g. [SK98]) have proposed to use *hash chains*, where each log entry is augmented by the hash value of the previous log message, which in turn contains the hash value of the previous log message, and so on. This detects reordering attacks as well as deletions of log entries, except from the end of the log file (truncation attacks).

- Instead of using hash chains, some newer proposals [CW09; Bul+14; RFC6962] utilize Merkle hash trees [Mer88] to combine hash values of distinct log records. These have additional structure (compared to hash chains), that can be used to provide additional features such as efficient "proofs of inclusion" (i.e. efficient proofs that a given log record is contained in the log).

- Some schemes add *epoch markers* to the log file. An epoch marker is a dedicated log record conveying the information that an epoch has ended and the secret key is being updated. A verifier can then determine which key index i to use for verifying a log entry by counting the number of epoch markers before the log entry. [BY97]

- If a scheme performs epoch switches independently of the amount of time passed since the last epoch switch, it may be sensible to just add log entries containing the current time in regular intervals. Such log entries are called *metronome entries*. [Hol06]

- Some schemes additionally employ encryption to protect the confidentiality of log messages, e.g. [SK98; Hol06; Bul+14].

1.2.2. Overview of Individual Publications

We now review several schemes and publications concerning the secure storage of log files in more detail. The schemes are given in chronological order. While this review can in no way claim completeness, it highlights the basis on which the research presented in this thesis is built. The reader might want to consider a somewhat dated survey paper [Acc09] on secure logging in addition to the overview given below.

The oldest mentioning of protocols to protect the integrity of log files appears to be due to Futoransky and Kargieman [FK95; FK98], but passed mostly unnoticed. They built various schemes based on hash chains, with a secret start value which functions as an initial secret key.

The study of cryptographic mechanisms to protect log files has been brought to wider attention by Bellare and Yee [BY97] in 1997. Motivated by the task to verify the operation of an initially trusted machine in an untrusted and potentially adversarial environment, they introduced the notion of forward integrity for MAC schemes.

They developed a simple scheme of forward-secure MACs (based on a key-chain generated by a pseudorandom function) and augmented that scheme with sequence numbers and epoch markers to add protection against the deletion of individual log entries.

Schneier and Kelsey [SK98; SK99] devised a more concrete scheme for secure logging using MACs. The MAC key is continuously evolved using a hash function, similar to Bellare and Yee's scheme. Schneier and Kelsey assume an untrusted machine U collecting the log entries, a trusted machine T that holds the initial MAC key (and thus can verify the complete log) and a semi-trusted log verifier V. Their scheme includes encryption of log entries and a mechanism for T to grant the semi-trusted verifier V read access to individual log entries.

Waters et al. [Wat+04] focused on encryption of log entries in a way that allows for efficient keyword-search in the log file. They do not develop new techniques to guarantee log file integrity. Therefore, their line of research is mostly orthogonal to the one pursued in this thesis.

Xu, Chadwick, and Otenko [XCO05] built a web service for secure audit logs. Building on Schneier's and Kelsey's scheme, they proposed to (optionally) use public-key encryption and digital signatures instead of symmetric encryption and MACs. While this change decreases performance, it allows for publicly verifiable log files, since the verification key can be made public. In addition, they used trusted platform modules to store cryptographic keys.

Kiltz et al. [Kil+05] designed "append-only signatures", a type of signature scheme where given a signature (under some key pk) for a sequence of symbols $m = (m_1, \ldots, m_n)$, anyone should be able to append to m, and produce a valid signature (under pk, too) for the resulting sequence. Kiltz et al. showed that this notion is equivalent to hierarchical identity-based signature schemes [GS02]. However, Kiltz et al. aim for different applications than ours, and hence they do not consider forward security. Moreover, a scheme where anyone (instead of only the legitimate logger) can append to such sequences and produce a valid signature for the resulting sequence would violate our expectations of security in the log signing case.

Kawaguchi et al. [Kaw+05] proposed a logging scheme where multiple hosts first arrange their log records in a Merkle tree locally, and then the root hashes of all hosts are communicated over network and arranged in another Merkle tree. The root node of the latter Merkle tree is then authenticated by an external, trusted Time Stamping Authority. The resulting signature is sent back to each host, along with the root hash values of other hosts, as necessary to verify the top Merkle tree.

Building on Schneier and Kelsey's scheme, Holt [Hol06] designed Logcrypt. Holt used a construction similar to the Schneier-Kelsey scheme, but (as [XCO05]) proposed to substitute digital signatures for the MACs used by Schneier and Kelsey. He implicitly constructed forward-secure signatures from standard signature schemes for his logging scheme.

Ma and Tsudik [MT08; MT09] showed that Schneier's and Kelsey's semi-trusted verifier V can easily be tricked into accepting a modified log file. This was termed a "delayed detection attack", since the fully trusted verifier T can indeed detect such

tampering, but is considered to check the log file at a later point in time. Moreover, Ma and Tsudik showed a truncation attack on the previous schemes, where the attacker deletes one or more log entries from the end of the log file.[6]

In response to these attacks, Ma and Tsudik proposed using "forward secure sequential aggregate" signatures (FssAgg signatures) [MT07b], which are a combination of forward-secure signatures with *sequential aggregate signatures*. As noted above, sequential aggregate signature schemes [Lys+04] are a special kind of signature schemes, where a single signature can simultaneously authenticate multiple messages. (In contrast, standard signature schemes require one signature per message.) Therefore, sequential aggregate signatures offer an attractive option to reduce storage and bandwidth overhead in applications requiring authentication of a large number of messages.

Since FssAgg schemes are public-key primitives, the verification key can be given to any verifier, preventing delayed detection attacks. Moreover, since only one (aggregated) signature needs to be kept in order to verify the log file, truncation attacks can be detected, as long as the attacker cannot "deaggregate" signatures for log entries from the aggregate signature (i.e. derive a signature for a *prefix* of a message sequence from a signature for the entire sequence).

While providing a single aggregate signature for the complete log file may help to avert truncation attacks, it also eliminates the possibility to check the integrity of individual log entries without checking the entire log file. In order to re-enable the verifier to do so, Ma and Tsudik modified their scheme to include an individual signature for each log entry as well as an aggregated signature for all log entries. This forced them to reconsider the deaggregation problem and strengthen their security notion to so-called "immutable" forward-secure sequential aggregate signatures, which offer some protection against deaggregation.

Later on, Ma [Ma08] devised two more FssAgg signature schemes (called BM-FssAgg and AR-FssAgg), which offer different tradeoffs in efficiency and build on other hardness assumptions. These schemes are sequential aggregate variants of forward-secure signature schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00]. This thesis presents a successful cryptanalysis of both sequential aggregate schemes in Chapter 3.

Crosby and Wallach [CW09] proposed a scheme where log records are not arranged in a hash chain (as with e.g. Schneier's and Kelsey's scheme [SK98; SK99]), but in a Merkle hash tree [Mer88] instead. Their scheme allows for the generation of excerpts as well as controlled deletion of certain log entries, while keeping the remaining log entries verifiable. In order to obtain completeness of excerpts, i.e. guarantees that no log entry has been omitted, they proposed to annotate tree nodes with attributes, which are propagated towards the root node. Thus, if a given node does not have a certain attribute, then none of its descendants will have that attribute. This allows for proving completeness in a reasonably efficient manner. However, their scheme relies on frequent communication between the log server and one or more trusted auditors that

[6] This truncation attack also applies to Logcrypt, which was already acknowledged in [Hol06]. Holt proposed to use metronome entries to deal with this issue.

need to store "commitments" to the log file, whereas this thesis strives to offer security in the standalone model.

In 2010, the Internet Engineering Task Force released a standard for "Signed Syslog Messages" [RFC5848]. This standard extends the syslog protocol, a widely used mechanism to collect and transmit log records in UNIX systems. The standard focuses on the secure transmission of log messages over a potentially untrusted and/or unreliable network, adding "origin authentication, message integrity, replay resistance, message sequencing, and detection of missing messages to syslog" [RFC5848, Section 1].

In this standard, messages are collected in "signature groups" of limited size. Newly arriving log messages are assigned to the respective group. The *signer* (who can be the originator of the log messages or an intermediary) will eventually sign the entire signature group, send the signature to the receiver, and then close this signature group for the addition of more log records. Log messages arriving in the future are then added to another signature group, and the process repeats.

While the security requirements quoted above are essential for the *transmission* of log entries over a network, the techniques proposed by the standard offer only limited security for the *storage* of log entries as considered in this work. More specifically, the standard does not support forward integrity and therefore does not benefit the security of log messages captured in a fully standalone setting. Only in a setting where the messages are signed by one server before being sent to another server for storage does this approach offer protection for stored log entries, since an attacker will have to compromise both servers in order to manipulate log entries without risking detection.

In contrast, the security guarantees provided by the schemes developed in this thesis do *not* depend on the attacker's limited ability to compromise computer systems, but stand *regardless* of the number of entities compromised by the attacker, as long as the verifier remains uncompromised.

Driven by performance considerations on the signer side, Yavuz, Peng and Reiter [YP09; YPR12a] designed a scheme called "Blind-Aggregate-Forward" (BAF). While BAF has a very efficient signing procedure, the size of the public verification key is linear in the maximum number of supported epochs. While this is a sensible trade-off for applications where signers are subject to tight resource constraints (such as wireless sensors), it may be undesirable in other applications.

Another scheme by Yavuz, Peng and Reiter is LogFAS [YPR12b; YR11]. The verification algorithm for LogFAS requires less computational effort than BAF's verification algorithm, but the sizes of signing *and* verification keys for LogFAS are linear in the number of supported log entries.

LogFAS provides the option to extract signatures for single messages from a signature for the entire log file, and (by extension) provide signatures for excerpts from the log file. However, with LogFAS, any subsequence of the real log file can be verified (given the right signatures), and consequently, LogFAS can not guarantee completeness of the generated excerpt. This thesis presents two attacks on LogFAS in Chapter 3.

Marson and Poettering [MP13] devised "Seekable Sequential Key Generators" for a secure logging scenario (using MACs). These "SSKGs" basically form a hash chain based on a one-way function, where one can efficiently "seek forward", i.e. given the

i-th element in the chain e_i, one can quickly compute the j-th element e_j for each $j \geq i$ without having to evaluate the one-way function $j - i$ times. This can be useful if the verifier only needs to verify some of the log entries in a log file. They build such a "seekable" chain by using squaring modulo a Blum integer $N = PQ$, i.e. each element in the chain is the square of its predecessor. Given the factorisation (P, Q) of N, one can seek forward by first computing $f \coloneqq 2^{j-i} \mod \phi(N)$, where $\phi(N) = (P-1)(Q-1)$, and then calculating $e_j \coloneqq e_i^f = e_i^{(2^{j-i})} \pmod N$.

Certificate Transparency [RFC6962] is an experimental approach to detect mis-issued cryptographic certificates binding public keys to identities. The core idea is to require all certification authorities to submit the certificates issued by them to public logs. In this scenario, cryptographic software should reject any certificate that is not contained in such a public log as invalid. The log is organized as a Merkle hash tree, so log servers can efficiently prove that a given certificate is contained in the log. The Certificate Transparency approach crucially relies on *monitors* and *auditors* to regularly check the log state at a given point in time for consistency with previously observed log states. Hence, the certificate transparency approach is not applicable in the standalone setting considered in this work.

PillarBox [Bow+14] is a logging system focusing on additional properties such as confidentiality of log entries and logging rules. The authors assume forward-secure MACs as a tool, and use interaction with other servers to obtain truncation security.

Buldas et al. [Bul+14] build a logging scheme based on a Merkle tree structure, adding random values to hashes of leaves in order to strengthen the confidentiality offered by the hash function. They propose to authenticate the root of the Merkle tree with a digital signature scheme, but require external time stamping services to obtain security against attackers breaking into the logging host.

Lindqvist [Lin17] built a logging scheme with membership tests based on Bloom filters. His work focuses on confidentiality of the log data. Information about log entries is placed in Bloom filters, whose bits are then arranged in a Merkle tree. The Bloom filter can (with good probability) be used to show that certain log data is *not* contained in the log file.

Pulls and Dahlberg [PD18] proposed "Steady", a logging system where the storage of log records is outsourced to a *relay*, e.g. an untrusted cloud provider. In their scheme, a "client" first collects log entries into blocks, then computes a Merkle tree hash of all log entries in a block, signs the root hash, optionally compresses and encrypts the log data, and finally uploads it to the relay. A trusted verifier can later verify the log file based on signatures created by the client.

1.3. Contribution

This thesis advances the state of the art as follows.

Attacks on Previously Published Schemes. Firstly, it points out a total of four serious vulnerabilities in three of the schemes introduced above. All three schemes

were published at notable and peer-reviewed conferences, and all of them have an accompanying security proof which should rule out any meaningful attack on the schemes. In fact, three of the four vulnerabilities described here stand in contradiction to the security properties claimed and supposedly proven by the respective authors, while the fourth one is outside the respective security model. This thesis points out some mistakes in the proofs, resolving this contradiction.

In particular, this thesis presents two attacks on LogFAS [YPR12b; YR11], one of them allowing for virtually arbitrary forgeries, and one of them allowing for confusion of legitimate signers. Both of these attacks are very simple, yet were apparently overlooked during the peer-review process.

The other two attacks presented here regard the BM-FssAgg scheme and the AR-FssAgg scheme by [Ma08]. These schemes are built on forward-secure signature schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00], respectively, but have been modified in order to add support for sequential aggregation of signatures. These modifications involved replacing values chosen randomly per signature by interdependent values that have a compact representation. The attacks presented in this thesis exploit this "de-randomization" to recover the secret key sk_t for some epoch t from t consecutive signatures. The resulting attacks are non-trivial and require doing linear algebra "in the exponent". See Chapter 3 for more details.

Truncation Security. Secondly, this thesis presents formal security notions which include truncation security for logging schemes. Most prior work did not formally model this security requirement, and hence authors could only give informal arguments for the truncation security of their schemes. To the best of the author's knowledge, the only work formally considering truncation security is LogFAS [YPR12b]. However, the security notion given in [YPR12b] allows for omissions of log entries: For example, after three queries to the signature oracle (for messages m_1, m_2, and m_3, respectively), a forgery of a signature for the log file (m_1, m_3) is considered trivial in their notion. In contrast, our definitions typically consider such an attack as non-trivial, and our security proofs rule out such forgeries.

Moreover, this thesis presents a new technique which can be used to achieve truncation security without relying on external auditors or designated hardware. The technique works by a) requiring the log signer to add epoch markers (see p. 11) to the log file, each indicating an evolution of the current secret key sk_i to sk_{i+1}, and, b) upon verification, requiring the signer to "prove" that (s)he knows the most recent secret key sk_t, where t can be derived from the number of epoch markers in the log file.

This technique is sufficient to guarantee the same level of truncation security as obtained in prior work. The logging schemes presented in this thesis use this approach, thereby illustrating its application. The schemes are proven secure with regard to our security notions, and therefore provably attain truncation security. The technique is discussed in more detail in Section 4.3, in "proximity" to our first construction using this approach.

Log Files with Verifiable Excerpts. Thirdly, this thesis proposes a cryptographic scheme [Har16a] for the protection of log files that can support the creation of *excerpts* from log files, while maintaining the public verifiability of these excerpts, see Chapter 4. Excerpts can be checked for *integrity and authenticity* (no log entry in the excerpt has been retroactively modified or injected by an attacker) as well as for *completeness* (no relevant log entry has been omitted from the excerpt). This scheme appears to be the only one published to date that provides forward integrity and completeness of excerpts in the standalone model:

While logging schemes based on Merkle hash trees (e.g. [CW09; Bul+14; RFC6962]) have the ability to prove that specific log entries are contained in the log, and (by extension) generate excerpts, [Bul+14] and [RFC6962] neither explicitly consider excerpts of more than one log entry, nor provide for completeness. While [CW09] consider completeness, their scheme requires constant interaction with auditors to guarantee security, and hence their scheme is not in the standalone model.

Another scheme that can support the creation of excerpts is LogFAS [YPR12b; YR11], which can provide signatures for arbitrary subsets of the log records contained in a log file. However, LogFAS does not have mechanisms for proving completeness of the generated excerpt—and does not even offer security against forgeries, as pointed out above.

Fault-Tolerance for Log Files built from Aggregate Signatures. Fourthly, this work introduces a technique that combines the reduction of signature storage overhead offered by sequential aggregate signatures with resilience to log file modifications (i.e. the ability to verify the integrity and authenticity of parts of the log file when other parts have been modified). Previous approaches either had storage overhead proportional to the number of stored log records [BY97; SK98; Hol06; YPR12b; MP13] or used techniques like Merkle trees (e.g. [CW09]) or aggregate signatures [MT08; Ma08; YPR12a] to form a compact proof of authenticity, introducing only *sub-linear* storage overhead.

However, the latter approaches cause some *fragility*: The modification of a single log entry will render the entire signature invalid, preventing the cryptographic verification of any part of the log file. However, being able to distinguish manipulated log entries from non-manipulated ones may be of importance for after-the-fact investigations. The thesis at hand addresses this issue by presenting a new technique providing a trade-off between storage overhead and *robustness*, i.e. the ability to tolerate some modifications to the log file while preserving the cryptographic verifiability of unmodified log entries. This robustness is achieved by the use of a special kind of sequential aggregate signatures (called *fault-tolerant* sequential aggregate signatures [Har+17b]), which contain some redundancy. The construction of fault-tolerant aggregate signatures [Har+16] makes use of combinatorial methods guaranteeing that if the number of errors is below a certain threshold, then there will be enough redundancy to identify and verify *all* non-modified log entries. See Chapter 5 for more details.

Summary. This thesis advances the state of the art with regard to providing security for computer log files in a number of ways: by analyzing the security of previously proposed schemes, by defining formal security notions capturing truncation security and proposing a new technique to achieve resistance against log truncations, by providing the first scheme in the standalone model where excerpts can be verified for completeness, and by describing the first scheme which can achieve some notion of robustness while being able to aggregate log record signatures.

1.4. Paper Overview

This thesis is based on the following publications:

[Har+17b] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann. "Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures". In: *Provable Security*. Ed. by T. Okamoto, Y. Yu, M. H. Au, and Y. Li. Lecture Notes in Computer Science 10592. Springer International Publishing, 2017, pp. 87–106. ISBN: 978-3-319-68637-0. DOI: 10.1007/978-3-319-68637-0_6.

[Har16a] G. Hartung. "Secure Audit Logs with Verifiable Excerpts". In: *Topics in Cryptology — CT-RSA 2016*. Ed. by K. Sako. Lecture Notes in Computer Science 9610. Springer, 2016, pp. 183–199. ISBN: 978-3-319-29484-1. DOI: 10.1007/978-3-319-29485-8_11.

[Har16b] G. Hartung. *Secure Audit Logs with Verifiable Excerpts – Full Version*. Cryptology ePrint Archive, Report 2016/283. https://eprint.iacr.org/2016/283. 2016.

[Har17] G. Hartung. "Attacks on Secure Logging Schemes". In: *Financial Cryptography and Data Security*. Ed. by A. Kiayias. Springer International Publishing, 2017, pp. 268–284. ISBN: 978-3-319-70972-7. DOI: 10.1007/978-3-319-70972-7_14.

The publication [Har16b] is the informally published full version of [Har16a]. The former publication has *not* been completely peer-reviewed, but its core contributions are also given in the latter publication, which has passed peer review.

A significant fraction of the text contained in this thesis is copied—without or with only slight modifications—from the publications listed above. In order to keep this thesis readable, text passages taken from the former publications are not enclosed in quotes, and changes are not marked by enclosing them in brackets. See Appendix C for notes regarding the copyright of these publications.

Moreover, this thesis is indirectly based on [Har+16] (see below), since [Har+17b] is based on the former publication.

In addition, the author has participated in the research that has led to the following peer-reviewed publications, which do not form a part of this thesis:

[Bro+17] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. "Concurrently Composable Security with Shielded Super-Polynomial Simulators". In: *Advances in Cryptology – EUROCRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer International Publishing, 2017, pp. 351–381. ISBN: 978-3-319-56620-7. DOI: 10.1007/978-3-319-56620-7_13.

[Har+16] G. Hartung, B. Kaidel, A. Koch, J. Koch, and A. Rupp. "Fault-Tolerant Aggregate Signatures". In: *Public-Key Cryptography — PKC 2016*. Ed. by C. Cheng, K. Chung, G. Persiano, and B. Yang. Lecture Notes in Computer Science 9614. Springer, 2016, pp. 331–356. ISBN: 978-3-662-49383-0. DOI: 10.1007/978-3-662-49384-7_13.

[Har+17a] G. Hartung, M. Hoffmann, M. Nagel, and A. Rupp. "BBA+: Improving the Security and Applicability of Privacy-Preserving Point Collection". In: *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017, pp. 1925–1942. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134071.

2. Preliminaries

We begin by briefly introducing some preliminaries for this thesis. Preliminaries only relevant for a specific chapter of this thesis will be introduced in the respective chapter.

The content of this chapter is mostly a selection of the preliminaries for the research papers underlying this thesis [Har+16; Har16a; Har+17b] and the author's master's thesis [Har13]. Parts of this chapter have been taken from these publications without or with only minor modifications, and without further designation.

2.1. Notation and Basic Definitions

We start with some notations and definitions. It is assumed the reader is familiar with the concepts introduced here.

Notation 2.1 (Iff)**.** We use "iff" as an abbreviation for "if and only if", verbally expressing logical equivalence of two statements, where each statement implies the other.

Notation 2.2 (Definitions and Assignments)**.** We write $:=$ to denote mathematical definitions as well as assignment operations in algorithms. The symbol $\leftarrow$ is used to denote the assignment operation when we want to emphasize the probabilistic nature of the assignment, e.g. when assigning the output of a probabilistic algorithm to a variable.

If S is a finite, non-empty set, we also write $V \leftarrow S$ to indicate that the variable V is assigned a random value according to the uniform distribution on S, i.e. for each $e \in S$: $\Pr[V = e] = \frac{1}{|S|}$. All random choices are considered to be independent, unless explicitly noted otherwise.

Notation 2.3 (Sets)**.** The set of natural numbers $\{1, 2, \ldots\}$ is denoted by $\mathbb{N}$, $\mathbb{N}_0$ is defined as $\{0\} \cup \mathbb{N}$. For a natural number $n \in \mathbb{N}$, we let $[n] := \{1, 2, \ldots, n\}$. The set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ is written as $\mathbb{Z}$. For $a, b \in \mathbb{Z}$, $\gcd(a, b)$ refers to the greatest common divisor of a and b, i.e. the greatest integer d such that d divides both a and b.

$\mathbb{R}$ is the set of real numbers, and $\mathbb{R}_0^+$ is the set of non-negative real numbers. For $a, b \in \mathbb{R}$, $[a, b]$ denotes the closed interval from a to b, i.e. $[a, b] := \{x \in \mathbb{R} \colon a \leq x \leq b\}$.

Notation 2.4 (String Literals)**.** String literals used by algorithms are indicated by typewriter text between quotation marks, e.g. "`End of epoch`".

Notation 2.5 (Modulo Arithmetic)**.** Let $a, b, N \in \mathbb{Z}$ and $N \geq 2$. $a \mod N$ is the remainder of the integer division a/N. We write $a \equiv b \pmod{N}$ iff $a \mod N = b \mod N$.

$\mathbb{Z}_N \coloneqq \mathbb{Z}/N\mathbb{Z}$ refers to the set of integers modulo N, which forms a commutative ring with one under the modular addition and modular multiplication operations. We do not distinguish operations on $\mathbb{Z}_N$ from operations on $\mathbb{Z}$ or $\mathbb{R}$ in our notation, and use the standard symbols $+$, $-$, $\cdot$, etc. to represent calculations in $\mathbb{Z}_N$. We may write integers for the elements of $\mathbb{Z}_N$, these integers are implicitly mapped to their respective residue class. Similarly, we may use residue classes from $\mathbb{Z}_N$ as integers, these are implicitly mapped to their "canonical" representative in $\{0, \ldots, N-1\}$. We may use $=$ to denote equality within $\mathbb{Z}_N$.

$\mathbb{Z}_N^*$ refers to the units of $\mathbb{Z}_N$, i.e. the elements $i \in \mathbb{Z}_N$ with $\gcd(i, N) = 1$. If N is a prime number, then $\mathbb{Z}_N$ is a finite field and is also written as $\mathbb{F}_N$. In this case, $\mathbb{Z}_N^* = \{1, \ldots, N-1\}$.

Notation 2.6 (Statements for almost all n)**.** We say that a statement is true for almost all $n \in \mathbb{N}$ if there exists a $n_0 \in \mathbb{N}$ such that the statement is true for all $n > n_0$.

Notation 2.7 (Asymptotic Behaviour of Functions)**.** For two functions $f, g : \mathbb{N} \to \mathbb{R}_0^+$ we write $f \in \mathcal{O}(g)$ iff there is a constant $c_{f,max} \in \mathbb{R}_0^+$ such that $f(n) \leq c_{f,max} \cdot g(n)$ for almost all n. $\mathcal{O}(g)$ is the set of all such functions. We write $f \in \Omega(g)$ iff a constant $c_{f,min} \in \mathbb{R}^+$ with $f(n) \geq c_{f,min} \cdot g(n)$ for almost all n exists. We let $\Theta(g) \coloneqq \Omega(g) \cap \mathcal{O}(g)$.

If $g(n)$ can be described by a closed formula, we usually simply write the formula instead of explicitly defining g. E.g., if $g(n) = n$, we simply write $\mathcal{O}(n), \Omega(n)$ and $\Theta(n)$ for $\mathcal{O}(g)$, $\Omega(g)$ and $\Theta(g)$, respectively.

Notation 2.8 (Polynomially Bounded Functions)**.** We write $a \in \mathrm{poly}(x)$ to indicate that a is a function of x and $a \in \mathcal{O}(x^c)$ for a constant $c \in \mathbb{N}$.

Notation 2.9 (Security Parameter)**.** Throughout this thesis, $\kappa \in \mathbb{N}$ denotes the security parameter. All algorithms are implicitly given the security parameter encoded in unary (written as 1^κ) as their first input, even when not explicitly denoted.

Notation 2.10 (PPT Algorithms)**.** We say an algorithm A is probabilistic polynomial time (PPT) if A is a probabilistic algorithm and the running time of A is upper-bounded by some polynomial $t \in \mathrm{poly}(\kappa)$.

Definition 2.11 (Negligible and Overwhelming Functions)**.** A function $f : \mathbb{N} \to \mathbb{R}_0^+$ is called negligible iff for each polynomial $p : \mathbb{N} \to \mathbb{R}_0^+$ it holds that $f(n) < 1/p(n)$ for almost all $n \in \mathbb{N}$. We also write $f(n) \leq \mathsf{negl}(n)$ to indicate that $f(n)$ is negligible. A function $g : \mathbb{N} \to [0, 1]$ is called overwhelming iff $1 - g(n)$ is negligible.

Notation 2.12 (Matrices)**.** For a non-empty set S, and $n, m \in \mathbb{N}$, $S^{n \times m}$ is the set of all matrices with n rows, m columns and entries in S. For $M \in S^{n \times m}$, $\mathrm{rows}(M) \coloneqq n$ and $\mathrm{cols}(M) \coloneqq m$ denote its number of rows and columns. $M[i, j]$ is the entry in the i-th row and j-th column of M.

Addition and multiplication of matrices is defined in the standard sense if S is a ring.

2.2. Sequences

In this thesis, we will constantly (implicitly or explicitly) be working with sequences. It is therefore in order to properly introduce sequences and their notation.

Notation 2.13 (Sequences)**.** If D is a non-empty set, D^* is the set of all finite sequences over D. Such sequences are denoted by parentheses, e.g. $(1, 2, 3)$. In particular, $\{0, 1\}^*$ is the set of all bit strings. The empty sequence is $()$.

A sequence $S = (s_1, \ldots, s_l)$ may also be written as $(s_i)_{i=1}^{l}$. The *length* of a sequence S is the number of (not necessarily distinct) elements in the sequence, and is denoted by $\text{len}(S) = l$. For an $i \in [\text{len}(S)]$, $S[i]$ refers to the i-th element in the sequence. We write $v \in S$ to indicate that some value v is contained in S, i.e., there exists an $i \in [l]$ such that $v = S[i]$.

The concatenation of two finite sequences S_1, S_2 over the same domain D is denoted as $S_1 \parallel S_2$. If $s \in D$ is a single element, we write $S_1 \parallel s$ as a shorthand for $S_1 \parallel (s)$.

Given an element $s \in D$ and a non-negative integer $n \in \mathbb{N}_0$, s^n is the sequence that results from repeating s for n times, i.e. $s^n \coloneqq (s_1, \ldots, s_n)$ where $s_1 = \ldots = s_n = s$.

The notation s^n may introduce some ambiguity in notation if multiplication on D is defined. We use this notation despite the arising ambiguity, as is standard practice in the literature. How to interpret this notation will usually be clear from the context. In particular, when we write 1^n or 0^n, we usually refer to the sequence notation introduced here.

For this work, we will often need to work with subsequences, which are defined next.

Definition 2.14 (Subsequences)**.** Let $S = (s_1, \ldots, s_l)$ be a sequence. If $I = (i_1, \ldots, i_n)$ is a (possibly empty) finite, strictly increasing sequence of numbers $i_j \in [l]$ (for all $j \in [n]$, with $n \in \mathbb{N}_0, n \leq l$), we call I an *index sequence for* S. $S[I] \coloneqq (s_{i_1}, \ldots, s_{i_n})$ is the *subsequence of* S *induced by* I.

Definition 2.15 (Prefixes)**.** If $S = (s_1, \ldots, s_l)$ is a sequence of length $l \in \mathbb{N}_0$ and $P = (s_1, \ldots, s_m)$ for some $m \leq l$, then P is a *prefix* of S.

Log messages and log files are special cases of sequences:

Definition 2.16 (Log Messages and Log Files)**.** A (plain) *log entry*, *log message* or *log record* m is a bit string, i.e. $m \in \{0, 1\}^*$. A (plain) *log file* is a finite, possibly empty sequence of log entries $M = (m_1, \ldots, m_l)$.[1]

Additional definitions regarding sequences will be introduced in the following chapters, when they are needed.

[1] Note that $M = (m_1, \ldots, m_l) \neq m_1 \parallel \ldots \parallel m_l$, i.e. we consider the log entries in M to be distinguishable.

2.3. Digital Signature Schemes

This thesis makes use of two specific types of digital signature schemes, namely *forward-secure* signature schemes, both in a "plain" and a *sequential aggregate* version. This section formally introduces these types of signature schemes and gives security notions for such schemes. The reader is referred to Section 1.1 for intuition on these types of signature schemes.

Since this thesis only makes black-box use of such signature schemes, our focus is on introducing the abstraction of such schemes, not on introducing specific constructions. Nonetheless, we briefly describe possible constructions in order to provide some more intuition and to illustrate our definitions.

Claims and Claim Sequences. In order to simplify working with lots of messages, public keys and epochs, we use the concept of *claims*, as introduced by [Har+16]. In our context, a claim is a tuple $c = (\mathsf{pk}, t, m)$, consisting of a public key pk, an epoch number $t \in \mathbb{N}$ and a message $m \in \{0,1\}^*$. It represents the to-be-proven proposition that the owner of the public key pk has authenticated the message m during epoch t. In this sense, a valid signature σ is a "proof" of this statement. Informally, a signature σ is valid for a claim c if the verification algorithm of a given signature scheme outputs 1 when given c and σ as input. When the public key pk and the epoch number t are clear from the context, we may more briefly say that σ is valid for m.

A *claim sequence* is simply a finite sequence of claims. In the context of (forward-secure) sequential aggregate signatures, an aggregate signature may authenticate an entire claim sequence (i.e. it proves all the claims in the claim sequence). If so, the signature is *valid* for the claim sequence.

Claim sequences may be empty. We assume there is an "empty signature" λ which is valid for the empty claim sequence. This empty signature can be trivial, i.e. a constant value such as 1, since there is nothing to be proven.

Security Notions. The security of digital signature schemes is usually defined via imaginary "experiments". In these experiments, an attacker, modeled as a probabilistic algorithm $\mathcal{A}$, is tasked to forge a signature, under the general conditions set out by the experiment setup. For example, the experiment setup defines the resources and utilities the adversary may use to forge a signature, and may restrict the attacker's freedom of action.

A digital signature scheme is considered *secure* with respect to the given experiment if all "efficient" adversaries have only negligible probability of successfully forging a signature. These experiments are sometimes viewed as "games" where a "challenger" plays the role of the experiment, and it is the adversaries' goal to beat the challenger by forging a valid signature.

The security experiments for signature schemes commonly allow the attacker $\mathcal{A}$ to obtain valid, honestly created signatures for messages chosen by $\mathcal{A}$. If so, the experiments require $\mathcal{A}$ to forge a signature σ^* for a message m^* for which $\mathcal{A}$ did not obtain a signature before. If $\mathcal{A}$ outputs a signature for some message m which was

honestly signed before, this forgery is considered *trivial*, and $\mathcal{A}$ is considered to be unsuccessful in this case.[2]

Correctness, Valid and Regular Signatures. Intuitively, security requires that only signatures created honestly by the owner of some given secret key are accepted by the verification algorithm. The converse property requires that the verification algorithm accepts all signatures created honestly by the correct use of the signing algorithm. This property is called *correctness*. Informally speaking, signatures honestly created by the use of the respective algorithms in the intended way are called *regular*. We will give precise definitions of being *regular* below. These formal definitions differ for (plain) forward-secure signatures and forward-secure sequential aggregate signatures, but the intuition is the same in both cases: Signatures are regular if they are created by using the digital signature scheme's algorithms in the intended way. Given this definition, we may rephrase correctness as the requirement that all regular signatures are valid.

The precise definitions of "valid" signatures, "regular" signatures, "security" and some other properties defined in this thesis differ for the various types of signature schemes and logging schemes presented in this thesis. For simplicity, we will refer to these definitions by the name of the property only.

For example, if Σ is a forward-secure signature scheme and σ is a signature created using Σ, then saying that σ is "valid" refers to the definition of valid signatures for forward-secure signature schemes (i.e. Definition 2.18 below). If Σ instead was a forward-secure sequential aggregate signature scheme, then the phrase "σ is valid" would refer to the notion of valid signatures for this type of signature scheme instead, i.e. Definition 2.23 (see below). Which of the respective definitions is meant at a given point in this thesis will hence be clear from the context.

2.3.1. Forward-Secure Signature Schemes

A forward-secure signature scheme [BM99] uses distinct secret keys for signing in each time interval (epoch). For efficiency reasons, schemes where each secret key can be computed from the previous one, and where there is only single, compact key for verification are desirable. However, these properties are not strictly required. Throughout this thesis we assume w.l.o.g. that the current epoch number can be efficiently derived from the current secret key.

Next, we will give formal definitions for forward-secure signature schemes. As is customary in cryptographic literature, we separate the syntax of signature schemes from their security notion, and consequently give separate definitions for these concepts. In the context of forward-secure signature schemes, we will define *key-evolving* digital signature schemes to denote the syntactical difference from standard signature schemes, and define a security notion capturing the property of forward security later.

We now briefly introduce the main differences between the syntax of standard signature schemes and key-evolving schemes. The latter ones have an additional

[2] This restriction is relaxed for the notion of *strong unforgeability* [ADR02], where $\mathcal{A}$ may additionally output modified signatures σ^* for messages m for which $\mathcal{A}$ obtained a signature σ.

algorithm called Update for key evolution. This algorithm takes a secret key sk_t as input and returns its successor sk_{t+1}. The verification algorithm is adapted canonically: Its input consists of a claim (containing a public key pk, an epoch number t and a message m) and a signature. In comparison, a claim for a standard signature scheme only consists of a public key and a message.

We do not require forward-secure schemes to support an unbounded number of epochs. Rather, the user must specify an upper bound T on the number of epochs when generating a new key pair. This bound is given as an input to the key generation algorithm, which must generate a key pair fit for use for at least T epochs. (However, a user may decide to stop using such a key-pair before the T-th epoch has passed, e.g. when the key is revoked.)

Definition 2.17 (Key-Evolving Signature Scheme, based on [BM99])**.** A *key-evolving signature scheme* is a tuple $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ of PPT algorithms, where

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk})$
: receives the security parameter κ and an a-priori upper bound T on the number of epochs as input. Both inputs are encoded in unary. It generates and outputs a pair of keys, consisting of the initial private signing key sk_1 and the public verification key pk.

$\mathsf{Update}(\mathsf{sk}_t) \to \mathsf{sk}_{t+1}$
: takes the secret key sk_t of period t as input. If $t \geq T$, the behaviour of Update may be undefined. Otherwise, Update evolves sk_t to sk_{t+1}, deletes sk_t in an unrecoverable fashion, and outputs sk_{t+1}.

$\mathsf{Sign}(\mathsf{sk}_t, m) \to \sigma$
: takes as input a secret key sk_t and a message $m \in \{0,1\}^*$ and outputs a signature σ for the claim (pk, t, m), where t is the epoch of sk_t and pk is the public key for sk_t.

$\mathsf{Verify}((\mathsf{pk}, t, m), \sigma) \to 0/1$
: checks if σ is a valid signature under public key pk for a given message m, supposedly signed with the secret key for the epoch t. It outputs 1 iff the signature is deemed valid, otherwise it outputs 0.

A key-evolving signature scheme is required to be *correct* as defined below.

Definition 2.18 (Valid and Regular Signatures)**.** Let $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ be a tuple of algorithms as defined above. Given a signature σ and a claim $c = (\mathsf{pk}, t, m)$, σ is called *valid* for c iff $\mathsf{Verify}(c, \sigma)$ outputs 1.

Moreover, σ is called *regular* for c iff it is in the image of $\mathsf{Sign}(\mathsf{sk}_t, m)$ where $T \in \mathrm{poly}(\kappa)$, $t \in [T]$, $(\mathsf{sk}_1, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$ and $\mathsf{sk}_{i+1} := \mathsf{Update}(\mathsf{sk}_i)$ for all $i \in [t-1]$.

Definition 2.19 (Correctness)**.** Let $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ be a tuple as defined above. FS is *correct* iff for all $\kappa \in \mathbb{N}$, all claims $c = (\mathsf{pk}, t, m)$, and all signatures σ regular for c, we have that σ is valid for c.[3]

In Definition 2.17, the maximum number of supported epochs T is encoded in unary for formal reasons: In many common constructions of forward-secure signature schemes (e.g. [BM99; AR00]), the running time of one or more algorithms may depend linearly on T. Hence, if T was given in binary, the algorithms had input length $\mathcal{O}(\log T)$ (assuming the security parameter κ is fixed), while the runtime is $\Omega(T)$. Thus, the algorithm's runtime would not be polynomial in the input length. We therefore need to encode T in unary.

In order to obtain a meaningful security notion, we will pass the maximum epoch number T (in unary) to the adversary as well, or otherwise the attacker might not even be able to execute the scheme's original algorithms. We thus need to demand that T be at most polynomial in the security parameter κ, or else the attacker may have super-polynomial runtime (in κ).

If an algorithm other than key generation has runtime linear in T, we may formally include 1^T in each secret and public key, respectively. This will make sure that the input length of all algorithms is at least $\Omega(T)$, and thus they may have runtime polynomial in T. However, in order to simplify notation, we will omit explicitly denoting 1^T as part of the keys. It is understood that 1^T must be contained in the secret and public keys output by the key generation and Update algorithms when the runtime of algorithms would not be polynomial in the input length otherwise.

The need to encode the parameter T in unary applies analogously to several other definitions in this work, e.g. Definitions 4.8 and 2.22.

Security Notion

The security notion for key-evolving signature schemes is mostly similar to the standard notion of *existential unforgeability under chosen message attacks*, but slightly more complicated, due to the presence of different epochs. It captures the "forward security" property, i.e. even if an adversary $\mathcal{A}$ knows the secret key of the current epoch, (s)he should not be able to forge a signature for any earlier epoch. In contrast, if $\mathcal{A}$ has obtained the secret key $\mathsf{sk}_{t_{\mathsf{BreakIn}}}$ of some epoch t_{BreakIn}, then $\mathcal{A}$ can trivially forge signatures for arbitrary messages m and epochs $t \geq t_{\mathsf{BreakIn}}$ by using the scheme's signing and updating algorithms.

The definition given below is roughly based on [BM99].

Definition 2.20 (Forward-Secure Existential Unforgeability under Chosen Message Attacks)**.** Let $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ be a key-evolving digital signature

[3] For certain key-evolving signature schemes FS, some of the algorithms constituting FS might have a small probability of failing. For example, an algorithm might fail to select two "large enough" prime numbers p, q by random sampling. In this case, the output of the respective algorithm is unclear. In order not to make the definitions in this work even more complicated, such issues are usually ignored here, as is customary in the literature.

scheme as defined above, $\mathcal{A}$ be a PPT algorithm with access to oracles as defined below, and $T := T(\kappa) \in \text{poly}(\kappa)$.

The security experiment for the notion of forward-secure existential unforgeability under chosen message attacks consists of the following phases:

Setup Phase.
The challenger creates a pair of keys $(\mathsf{sk}_1, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$ and initializes a counter $t := 1$. Afterwards $\mathcal{A}$ is called with inputs pk and 1^T.

Query Phase.
During the experiment, $\mathcal{A}$ may adaptively issue queries to the following two oracles:

Signature Oracle.
On input $m \in \{0,1\}^*$, the signature oracle computes the signature $\sigma = \mathsf{Sign}(\mathsf{sk}_t, m)$ for m using the current secret key sk_t. It returns σ to $\mathcal{A}$.

Epoch Switching Oracle.
Whenever $\mathcal{A}$ triggers the $\mathsf{NextEpoch}$ oracle, the experiment sets $\mathsf{sk}_{t+1} := \mathsf{Update}(\mathsf{sk}_t)$ and $t := t + 1$. The oracle returns the string "ok" to the adversary. $\mathcal{A}$ may invoke this oracle at most $T - 1$ times.

Break-In Phase.
When the adversary signals it is done with the query phase, the experiment switches to the break-in phase and queries to the signature and epoch switching oracles are no longer allowed.

During the break-in phase, the attacker may query a $\mathsf{BreakIn}$ oracle. If $\mathcal{A}$ does, the experiment sets $t_{\mathsf{BreakIn}} := t$ and returns sk_t to $\mathcal{A}$.

After $\mathcal{A}$ has invoked this oracle, it is no longer allowed any oracle queries.

Forgery Phase.
Finally, the attacker outputs a forgery (m^*, t^*, σ^*).

Let $t_{\mathsf{BreakIn}} := \infty$ if $\mathcal{A}$ did not query the $\mathsf{BreakIn}$ oracle. $\mathcal{A}$'s forgery (m^*, t^*, σ^*) is *trivial* iff $t^* \geq t_{\mathsf{BreakIn}}$ or $\mathcal{A}$ submitted the message m^* to the signature oracle during epoch t^*.

The experiment outputs 1 iff $\mathsf{Verify}((\mathsf{pk}, t^*, m^*), \sigma^*) = 1$ and (m^*, t^*, σ^*) is not trivial. Otherwise, the experiment outputs 0.

$\mathcal{A}$ is said to *win* an instance of the experiment defined above iff the experiment outputs 1. Otherwise, $\mathcal{A}$ *loses* the experiment.

A key-evolving signature scheme FS is *forward-secure existentially unforgeable under chosen message attacks* (or *FS-EUF-CMA-secure*) iff for each PPT adversary $\mathcal{A}$ and each $T \in \text{poly}(\kappa)$:

$$\Pr[\mathcal{A} \text{ wins}] \leq \mathsf{negl}(\kappa).$$

Remark 2.21. In the definition above, we have specified that the attacker may no longer use the signature oracle and epoch switching oracle once the break-in phase has begun. Note that this restriction is without loss of generality: Once the attacker has obtained the current secret key sk_t, (s)he can execute the signing and updating algorithms by him-/herself. (For the same reason, any forgery with regard to an epoch $t^* \geq t_{\mathsf{BreakIn}}$ is considered trivial.)

Constructions

We briefly point out how forward-secure signature schemes may be constructed.

Forward-secure digital signature schemes can be built from standard digital signature schemes in a black-box fashion. We explain some of these constructions along the lines of [MMM02], although some of these ideas already appeared in [BM99] and other previous works. Observe that we can consider any standard (non-key-evolving) digital signature scheme which is existentially unforgeable under chosen message attacks [GMR84; GMR88] as a key-evolving signature scheme that is fit for at most $T = 1$ epochs and secure according to the notion given above: If the attacker does not steal the secret key, then the experiment is equivalent to the standard existential unforgeability experiment, and if the attacker does break in, then all forgeries are trivial.

Malkin, Micciancio, and Miner [MMM02] formalized two methods of composing forward-secure signature schemes FS_1, FS_2 for T_1, T_2 epochs, respectively: These methods are called the "sum" composition and the "product" composition. We briefly summarize these composition methods, the reader is referred to [MMM02] for more detailed presentations. Let $(\mathsf{KeyGen}_i, \mathsf{Update}_i, \mathsf{Sign}_i, \mathsf{Verify}_i)$ be the algorithms of the respective scheme, for $i \in \{1, 2\}$.

Sum Composition. In the sum composition, one generates two key pairs $(\mathsf{sk}_1^1, \mathsf{pk}_1) \leftarrow \mathsf{KeyGen}_1(1^\kappa, 1^{T_1}), (\mathsf{sk}_2^1, \mathsf{pk}_2) \leftarrow \mathsf{KeyGen}_2(1^\kappa, 1^{T_2})$. One publishes $\mathsf{pk} = (\mathsf{pk}_1, T_1, \mathsf{pk}_2, T_2)$, and keeps $(\mathsf{sk}_1^1, \mathsf{sk}_2^1)$ as the initial secret key. For the first T_1 epochs, messages are signed with the respective key sk_1^t, and only sk_1 is updated. After T_1 epochs, the first secret key $\mathsf{sk}_1^{T_1}$ is deleted, and further epoch switches update sk_2 instead. New messages are signed with $\mathsf{sk}_2^{t-T_1}$. The resulting scheme supports $T = T_1 + T_2$ epochs, and is secure if FS_1 and FS_2 are forward-secure for T_1, T_2 epochs, respectively.

Product Composition. In the product composition, one uses a key-pair of FS_1 to "certify" public keys of the second scheme. In more detail, one initially generates $(\mathsf{sk}_1^1, \mathsf{pk}_1) \leftarrow \mathsf{KeyGen}_1(1^\kappa, 1^{T_1}), (\mathsf{sk}_2^1, \mathsf{pk}_2) \leftarrow \mathsf{KeyGen}_2(1^\kappa, 1^{T_2})$ and computes $\sigma_{\mathsf{pk}} := \mathsf{Sign}_1(\mathsf{sk}_1^1, \mathsf{pk}_2)$. Then sk_1^1 is updated to sk_1^2. The initial secret key is $(\mathsf{sk}_1^2, \mathsf{sk}_2^1, \mathsf{pk}_2, \sigma_{\mathsf{pk}})$, the public key is pk_1. For the first T_2 epochs, messages are signed with the respective sk_2^t of FS_2. The signature for a message m signed during epoch $t \in [T_2]$ is

$$\sigma := (\sigma' := \mathsf{Sign}_2(\mathsf{sk}_2^t, m), \mathsf{pk}_2, \sigma_{\mathsf{pk}}).$$

It is verified by checking if both $\mathsf{Verify}_1((\mathsf{pk}_1, 1, \mathsf{pk}_2), \sigma_{\mathsf{pk}})$ and $\mathsf{Verify}_2((\mathsf{pk}_2, t, m), \sigma')$ output 1.

After T_2 epochs, $\mathsf{sk}_2^{T_2}, \mathsf{pk}_2$ and σ_{pk} are deleted, and a new key-pair $(\mathsf{sk}_2^1, \mathsf{pk}_2) \leftarrow \mathsf{KeyGen}_2(1^\kappa, 1^{T_2})$ is generated. The new public key is certified by recomputing $\sigma_{\mathsf{pk}} := \mathsf{Sign}_1(\mathsf{sk}_1^2, \mathsf{pk}_2)$, and sk_1^2 is updated to sk_1^3. New messages m are signed by computing

$$\sigma := (\sigma' := \mathsf{Sign}_2(\mathsf{sk}_2^{t-1 \cdot T_2}, m), \mathsf{pk}_2, \sigma_{\mathsf{pk}}).$$

After another T_2 epochs, the "used up" key pair for FS_2 is deleted again, and replaced by a new key pair, which is again certified by the secret key of FS_1, and so on.

The result of this product composition is a scheme that can support $T = T_1 \cdot T_2$ epochs. The scheme is forward-secure if the underlying schemes $\mathsf{FS}_1, \mathsf{FS}_2$ are secure.

Tree-Based Schemes. Using these composition methods, there are various ways to construct forward-secure signature schemes for a high number of epochs T. We briefly present one possibility next.[4] One first uses the additive composition method to obtain a forward-secure signature scheme FS for $T = 2$ epochs from two standard signature schemes (which are considered as forward-secure signature schemes for 1 epoch). One then applies the product composition method $l - 1$ times ($l \in \mathbb{N}$) to this scheme to build a scheme for 2^l epochs.

The resulting scheme can be represented by a binary tree where each non-leaf node is associated with a pair of public keys. In each such node, the left key is used to sign the two public keys of the left child node, the right key is used to sign the keys of the right child node. The keys in the leaf nodes are used to sign the messages. New nodes are generated randomly on demand, their public keys are immediately authenticated by the corresponding key in the parent node, and the parent's secret key used for this authentication is then deleted.

The tree needs to have only l levels for a total of 2^l epochs, and can thus support an *exponential* number of epochs. If $T \in \mathrm{poly}(\kappa)$ instead, one only needs a tree of logarithmic height, and the scheme therefore only incurs logarithmic overhead.

A similar tree-based scheme can be built from hierarchical identity-based signatures [HWI03] (see also [GS02]).

Other Constructions. Besides these generic constructions, there exist a number of more direct constructions of forward-secure signature schemes, e.g. [BM99; AR00; IR01]. We do not give more detailed presentations of these schemes, but note that sequential aggregate versions of the schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00] will be presented in Sections 3.3.1 and 3.3.2. While these sequential aggregate variants are broken in Section 3.3, the original schemes [BM99; AR00] are unaffected by the attacks presented in Section 3.3.

2.3.2. Forward-Secure Sequential Aggregate Signatures

Sequential aggregate signatures were first introduced by Lysyanskaya et al. [Lys+04] as a restricted form of aggregate signatures [Bon+03]. The core idea of aggregate

[4] The scheme presented here differs from the scheme that Malkin, Micciancio, and Miner [MMM02] call the MMM scheme.

signatures is to "combine" signatures σ_i for individual messages m_i (valid under some public key pk_i) into a single signature σ which simultaneously authenticates all m_i under their respective pk_i.[5] The resulting signature σ is called the *aggregate* of all σ_i.

Aggregate signatures have the potential to greatly reduce storage and bandwidth overhead in applications dealing with large numbers of signatures, if the resulting aggregate signature σ is significantly "smaller" (in terms of required storage space) than the total space required to store all σ_i. Thus, the aggregate signature σ can be viewed as a "compressed form" of the signatures σ_i.

Consider, for example, the aggregate signature scheme by Boneh et al. [Bon+03]. In this scheme, signatures are elements of some group $\mathbb{G}$, and aggregating two signatures σ_1, σ_2 is realized by multiplying these signatures in the group, i.e. the aggregate signature is $\sigma = \sigma_1 \cdot \sigma_2$. Therefore, aggregation is very flexible: Anyone can aggregate signatures at any time in an ad-hoc fashion, signatures can be aggregated in any order, and aggregate signatures can be aggregated further.

Sequential aggregate signatures [Lys+04] are less flexible than general aggregate signatures, but can be constructed and proven secure (see e.g. [Lys+04]) without requiring the random oracle heuristic [BR93]. Whereas (general) aggregate signatures allow for ad-hoc aggregation of signatures created completely independently at any time and by anyone, sequential aggregate signatures only require that the signer can aggregate a signature with an already existing aggregate *at the time of signing*. Thus, if n signers wish to create an aggregate signature $\sigma_{1,n}$ for messages $m_1, \ldots, m_n$ (each signed by the i-th signer), they need to cooperate as follows: The first signer authenticates m_1 to obtain σ_1 and sends σ_1 to the second signer. The second signer computes a new aggregate signature $\sigma_{1,2}$ based on the signer's secret key, m_2 and σ_1.[6] The new signature simultaneously authenticates both m_1 and m_2. This signature is sent on to the third signer who uses it to create $\sigma_{1,3}$, valid for m_1, m_2, m_3 and so on.

In the context of this work, usually there will only be a single signer, eliminating the need for cooperation. However, the signer will use a signature scheme which not only supports sequential aggregation but also forward security (as explained above). Thus, the signer may use different secret keys sk_i to authenticate messages, and the forward-secure sequential aggregate signature scheme is expected to allow aggregation of signatures created with different sk_i, as well as to retain the information which message was signed during which epoch.

The following definition of key-evolving sequential aggregate signature (SAS) schemes is based on [Ma08], which in turn is based on [BM99] and [Lys+04]. It combines the forward security property [BM99] and the property of sequential aggregation [Lys+04].

[5] Aggregate signatures can be considered a generalization of multi-signatures, where different signers (with public keys pk_i) jointly authenticate a single message m. In contrast with multi-signatures, aggregate signatures lift the restriction that all signers authenticate the same message.

[6] How σ_1 is used during this process depends on the signature scheme at hand. One common way is to first create a signature σ_2 for the message m_2 and then multiply the signatures σ_1, σ_2 in order to obtain $\sigma_{1,2}$, as in the BGLS scheme. In fact, each fully flexible aggregate signature scheme can be viewed as a sequential aggregate signature scheme in this way.

It differs from the definition of key-evolving (non-aggregate) signature schemes in the following respects:

- Firstly, the signing algorithm Sign is replaced by an $\mathsf{AggSign}$ algorithm. This algorithm has two additional inputs: a signature-so-far σ_{i-1} and a claim sequence C_{i-1}, purportedly authenticated by σ_{i-1}. Its task is to somehow authenticate the message m_i in addition to the prior claim sequence C_{i-1}. The resulting signature σ_i should authenticate the claim sequence $C_i := C_{i-1} \| (\mathsf{pk}, t, m_i)$.
- Secondly, the verification algorithm now checks an entire sequence of claims (instead of a single claim). It outputs 1 if the signature is valid for the entire claim sequence C, and 0 otherwise.

Definition 2.22 (Key-Evolving Sequential Aggregate Signature Schemes)**.** A *key-evolving sequential aggregate signature* scheme is a tuple of four PPT algorithms $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$, where

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk})$
: takes as input the security parameter κ and an a priori upper bound T on the number of epochs. It generates and outputs a key pair $(\mathsf{sk}_1, \mathsf{pk})$, where sk_1 is the initial secret key for the first epoch, and pk is the public verification key.

$\mathsf{Update}(\mathsf{sk}_t) \to \mathsf{sk}_{t+1}$
: takes as input the secret key sk_t of period t. If $t \geq T$ the output of Update may be undefined. If $t < T$ it computes the secret key sk_{t+1} for the following period $t + 1$. It then securely erases the old secret key sk_t irrecoverably and outputs sk_{t+1}.

$\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i) \to \sigma_i$
: takes as input a secret key sk_t for an epoch t, a claim sequence C_{i-1}, a corresponding signature σ_{i-1} and a message m_i. It outputs a signature σ_i for the new claim sequence $C_i := C_{i-1} \| (\mathsf{pk}, t, m_i)$.

$\mathsf{Verify}(C, \sigma) \to 0/1$
: takes as input a claim sequence C and a signature σ and outputs 1 iff σ is deemed valid for C, and 0 otherwise.

Key-evolving sequential aggregate signature schemes are required to be *correct* as defined below.

Recall that we assume the existence of an empty signature λ which can be used for the empty claim sequence ().

Definition 2.23 (Valid and Regular Signatures)**.** Let $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$ be a tuple of PPT algorithms as defined above, C_i be a claim sequence and σ_i be a signature.

We say that σ_i is *valid* for C_i iff $\mathsf{Verify}(C_i, \sigma_i) = 1$. Moreover, σ_i is *regular* for C_i iff either

- $C_i = ()$ and $\sigma_i = \lambda$, or
- $C_i = C_{i-1} \| (\mathsf{pk}, t, m_i)$ for some claim (pk, t, m_i), $t \in [T]$, and σ_i is in the image of $\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$ where σ_{i-1} is a regular signature for C_{i-1}, m_i is an arbitrary message, $(\mathsf{sk}_1, \mathsf{pk})$ is a key pair output by $\mathsf{KeyGen}(1^\kappa, 1^T)$, and $\mathsf{sk}_{j+1} := \mathsf{Update}(\mathsf{sk}_j)$, for $j \in [T-1]$.

Definition 2.24 (Correctness)**.** Let AS be a tuple of algorithms as defined above. AS is *correct* iff for all $\kappa \in \mathbb{N}$, for all bounds on the number of epochs $T := T(\kappa) \in \mathrm{poly}(\kappa)$, and all claim sequences C, it holds that all signatures σ which are regular for C are also valid for C.

Note that in case of verification failure, a standard key-evolving SAS scheme provides no information on which claims are or aren't "true": The verification algorithm returns a single bit, indicating if either the signature is valid for the entire claim sequence (i.e. all claims in the sequence can be trusted), or the signature is invalid (i.e. at least one claim could not be confirmed as authentic). This "all-or-nothing" behavior of verification is relaxed in Section 5.3.

Security Notion for FS-SAS Schemes

The security notion for forward-secure sequential aggregate signatures requires that the information on which message was signed by which signer(s) during which epochs is retained in the aggregate signature, and this information is hard to forge. The security experiment combines the experiments of forward-secure signatures [BM99] and sequential aggregate signatures [Lys+04].

Definition 2.25 (Forward-Secure Sequential Aggregate Signature Existential Unforgeability under Chosen Message Attacks, based on [Ma08; Lys+04])**.** Let $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$ be a key-evolving sequential aggregate digital signature scheme as defined above, $\mathcal{A}$ be a PPT algorithm with access to oracles as defined below, and $T := T(\kappa) \in \mathrm{poly}(\kappa)$.

The security experiment for the notion of forward-secure sequential aggregate existential unforgeability under chosen message attacks consists of the following phases:

Setup Phase.

The challenger generates a key pair $(\mathsf{sk}_1, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$, where T is the maximum number of time periods. It initializes the epoch counter $t := 1$. The challenger then starts the adversary $\mathcal{A}$ with the inputs pk and 1^T.

Query Phase.

During the query phase, the adversary $\mathcal{A}$ has access to the following oracles, which may be queried adaptively:

Signature Oracle.

The $\mathsf{AggSign}$ oracle takes as input a claim sequence C_{i-1}, a corresponding signature σ_{i-1} and a message m_i. It computes and returns $\sigma_i :=$

$\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \sigma_{i-1}, m_i)$, where sk_t is the secret key for the current period t.

Epoch Switching Oracle.
When $\mathcal{A}$ calls the epoch switching oracle, the challenger computes $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t)$, increments $t \coloneqq t + 1$, and returns the string "ok". $\mathcal{A}$ is not permitted to query this oracle more than $T - 1$ times.

Break-In Phase.
When the adversary is done with the query phase, the experiment enters the Break-In phase.

The adversary may send a *break-in* request to obtain the current secret key by using a $\mathsf{BreakIn}$ oracle. The $\mathsf{BreakIn}$ oracle does not take parameters and returns sk_t to $\mathcal{A}$. If $\mathcal{A}$ queries the $\mathsf{BreakIn}$ oracle, the experiment sets $t_{\mathsf{BreakIn}} \coloneqq t$.

After $\mathcal{A}$ has used the $\mathsf{BreakIn}$ oracle, $\mathcal{A}$ may no longer query any of the oracles. Let $t_{\mathsf{BreakIn}} \coloneqq \infty$ if $\mathcal{A}$ does not query the $\mathsf{BreakIn}$ oracle.

Forgery Phase.
Finally, $\mathcal{A}$ outputs a claim sequence C^* and a corresponding signature σ^*.

We say that a claim $c^* = (\mathsf{pk}^*, t^*, m^*) \in C^*$ is *non-trivial* iff $\mathsf{pk}^* = \mathsf{pk}$, $t^* < t_{\mathsf{BreakIn}}$ and $\mathcal{A}$ did not query m^* at its $\mathsf{AggSign}$ oracle during epoch t^*.

The experiment outputs 1 iff $\mathsf{Verify}(C^*, \sigma^*) = 1$ and C^* contains a non-trivial claim. Otherwise, the experiment outputs 0.

The adversary $\mathcal{A}$ *wins* an instance of this experiment iff the instance outputs 1, otherwise $\mathcal{A}$ *loses* this instance.

A FS-SAS scheme AS is *forward-secure sequential aggregate signature existentially unforgeable under chosen message attacks* (*FS-SAS-EUF-CMA-secure*) iff for each $T \coloneqq T(\kappa) \in \mathrm{poly}(\kappa)$ and all PPT adversaries $\mathcal{A}$:

$$\Pr[\mathcal{A} \text{ wins}] \leq \mathsf{negl}(\kappa).$$

Note that $\mathcal{A}$ has much freedom in this experiment: The adversary may create any number of key pairs $(\mathsf{sk}_1, \mathsf{pk})$ for him-/herself (which also models that the attacker may corrupt other parties), or even use public keys created maliciously. The attacker can build claim sequences arbitrarily interleaving claims referring to the challenge public key pk^* and other public keys $\mathsf{pk} \neq \mathsf{pk}^*$. Additionally, $\mathcal{A}$ may arbitrarily branch claim sequences constructed in the experiment. For example, after building a claim sequence $C = (c_1, c_2)$ the attacker may build different continuations of C, for example $C' = (c_1, c_2, c_3)$ and $C'' = (c_1, c_2, c_4)$ for $c_3 \neq c_4$. If c_3 or c_4 refer to the challenge public key pk^*, the attacker can use the signature oracle to obtain signatures for the respective claim sequence(s). Moreover, the attacker is free to build several independent claim sequences in parallel, which may or may not overlap.

Note that some security notions found in the literature impose restrictions on the use of (forward-secure) sequential aggregate signatures: For example, Ma [Ma08] proposes

two schemes which can only support a single signature per epoch. As another example, the security notion given by [Lys+04] (for sequential aggregate signatures without forward security) requires that all public keys in a given claim sequence are distinct. In contrast to these notions, the security notion given above does not introduce such restrictions. We note that the BGLS-FssAgg scheme by [MT07b] (briefly introduced below) can be shown to be secure according to the notion defined above.

There is another difference between the security notion given above and some notions from the literature: The notions by [Lys+04; MT07b] consider the mere reordering of claims in a claim sequence as a non-trivial attack, whereas such a forgery is considered trivial in our notion. We stress that our security notions for keeping log files *will* consider the order of messages.

Constructions

As an example construction, we briefly point out the BGLS-FssAgg scheme by [MT07b], which is based on the BGLS aggregate signature scheme [Bon+03]. The BGLS-FssAgg scheme (like the BGLS scheme) uses a cryptographic pairing. A (type 3) *pairing* is a function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ mapping two elements from finite cyclic groups $\mathbb{G}_1$, $\mathbb{G}_2$ of order $q \in \mathbb{N}$ to an element in another finite cyclic group $\mathbb{G}_3$ (also of order q), where:

- e is *non-degenerate*, i.e. for all generators g_1, g_2 of $\mathbb{G}_1$, $\mathbb{G}_2$, respectively, $e(g_1, g_2)$ is a generator of $\mathbb{G}_3$,
- e is *bilinear*, i.e. for all group elements $a \in \mathbb{G}_1$, $c \in \mathbb{G}_2$, and all exponents $x, y \in \mathbb{Z}_q$, we have that $e(a^x, c^y) = e(a, c)^{xy}$,
- e can be computed efficiently, and
- there are no known non-trivial, efficiently computable homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$.

Since the groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_3$ are assumed to be cyclic, the bilinearity condition implies that for all $a, b \in \mathbb{G}_1$, $c, d \in \mathbb{G}_2$:

$$\begin{aligned} e(a \cdot b,\ \ c\ \ ) &= e(a, c) \cdot e(b, c), \text{ and} \\ e(\ \ a\ \ , c \cdot d) &= e(a, c) \cdot e(a, d). \end{aligned}$$

For the BGLS scheme, the group order q is usually chosen as a prime number.

Given a pairing and the corresponding groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_3$ with generators g_1, g_2, $g_3 = e(g_1, g_2)$, the BGLS-FssAgg scheme works as follows. For key generation, one selects a random exponent $x \leftarrow \mathbb{Z}_q$. The initial secret key is $\mathsf{sk}_1 := x$. Subsequent secret keys are computed as $\mathsf{sk}_{t+1} := H_1(\mathsf{sk}_t)$ for a hash function $H_1 : \mathbb{Z}_q \to \mathbb{Z}_q$ modelled as a random oracle [BR93] and $t \in [T-1]$. For each of these secret keys, one computes the public key for the respective epoch as $\mathsf{pk}_t := g_1^{\mathsf{sk}_t}$. The overall public key for a signer is $\mathsf{pk} := (\mathsf{pk}_1, \ldots, \mathsf{pk}_T)$. The update procedure simply replaces sk_t by $\mathsf{sk}_{t+1} := H_1(\mathsf{sk}_t)$, as above.

During epoch t, a message m is signed by computing $\sigma \coloneqq H_2(m)^{\mathsf{sk}_t}$, where $H_2 : \{0,1\}^* \to \mathbb{G}_2$ is another hash function, which is modeled as a random oracle, too.[7] For verification of a single claim $c = (\mathsf{pk}, t, m)$ one checks if

$$e(g_1, \sigma) \stackrel{?}{=} e(\mathsf{pk}_t, H_2(m)).$$

If σ is computed honestly, then this equation holds, since

$$\begin{aligned} e(g_1, \sigma) = e\Big(g_1, H_2(m)^{\mathsf{sk}_t}\Big) = e(g_1, H_2(m))^{\mathsf{sk}_t} \\ = e\Big(g_1^{\mathsf{sk}_t}, H_2(m)\Big) = e(\mathsf{pk}_t, H_2(m)). \end{aligned} \tag{2.1}$$

Signatures can be aggregated by multiplication, i.e. if $\sigma_{1,l-1}$ is a signature for a claim sequence C_{l-1} ($l \in \mathbb{N}$), and σ_l is a signature for a claim $c_l = (\mathsf{pk}_l = (\mathsf{pk}_{l,1}, \ldots, \mathsf{pk}_{l,T}), t_l, m_l)$, then $\sigma_{1,l} \coloneqq \sigma_{1,l-1} \cdot \sigma_l$ is the signature for $C_l \coloneqq C_{l-1} \,\|\, c_l$.

Such aggregate signatures are verified by checking

$$e(g_1, \sigma_{1,l}) \stackrel{?}{=} \prod_{i=1}^{l} e\Big(\mathsf{pk}_{i,t_i}, H_2(m_i)\Big).$$

Correctness can be shown by first expanding the left hand side as follows:

$$e(g_1, \sigma_{1,l}) = e\left(g_1, \prod_{i=1}^{l} \sigma_i\right) = \prod_{i=1}^{l} e(g_1, \sigma_i).$$

One then applies (2.1) analogously to each element in the product.

The security of the BGLS-FssAgg scheme can be shown under the computational co-Diffie-Hellman assumption (see [MT07b; Bon+03]) if H_1 and H_2 are modeled as random oracles.

This concludes our brief description of the BGLS-FssAgg scheme. The reader is referred to [MT07b] and [Bon+03] for more details. We will briefly present two more key-evolving sequential aggregate signature schemes in Sections 3.3.1 and 3.3.2, but these schemes are not secure as we will show in Section 3.3.

2.4. Conventions

We assume that we may use each signature scheme (with some message space $\mathcal{M}$) to sign certain mathematical objects o even if $o \notin \mathcal{M}$. (For example, we will need to sign tuples and partial maps $f : \{0,1\}^* \mapsto \mathbb{N}$ with finite support.) More precisely, we assume that whenever we sign some object o, then o is first encoded to a bit string $m \in \{0,1\}^*$ in some uniquely invertible encoding, then hashed with some collision

[7] The original proposal by Ma and Tsudik [MT07b] adds a counter i to m before signing in order to authenticate the order in which messages are signed. We will omit this counter here, since our security notion for sequential aggregate forward-secure signatures does not require the order of claims to be authenticated.

resistant hash function $H : \{0,1\}^* \to \mathcal{M}$, and then $H(m)$ is signed as a representative of o. Throughout this thesis, we assume that the encoding is injective, i.e. there are no two objects $o \neq p$ that are encoded to the same bit string. We employ the convention that the encoding and the hash function H are applied implicitly when invoking the signing and verification algorithms.

We do not model different "levels" of corruption of a system by an attacker. That is, we assume that when an attacker breaks into a system, this break-in is complete and the attacker immediately gains full control of the respective system. This assumption is in accordance with our goal to give security guarantees based on mathematical assumptions, instead of assumptions about the presence or absence of software vulnerabilities in the respective systems.

Throughout this thesis, we assume that key evolution and signing algorithms (for example Update, Sign or $\mathsf{AggSign}$) have access to the public key belonging to the respective secret key. As stated before, we also assume that the index t of a secret key sk_t can be extracted from sk_t efficiently.

For simplicity, we assume that all verification algorithms (i.e. Verify) are deterministic.

3. Attacks on Logging Schemes

This chapter presents a total of four attacks on three logging schemes proposed in the literature, namely the LogFAS scheme [YPR12b] and the BM- and AR-FssAgg schemes [Ma08].

This chapter is strongly based on [Har17]. Significant parts of that work are reproduced in this chapter without or with only minor modifications, and without specific designation.

3.1. Introduction

We describe two attacks on LogFAS [YPR12b; YR11] and one attack each on the BM- and AR-FssAgg schemes of [Ma08].

LogFAS. LogFAS assumes a "Key Generation Center" (KGC) which generates the secret keys and public keys for all signers. Signer-specific keys are derived from common secrets held by the KGC, the signers identity ID and per-signer values chosen at random.[1] Signatures consist of a number of group elements (of a prime-order subgroup of $\mathbb{Z}_p^*$ for a large prime p) and exponents for other group elements, as well as a standard signature on the log file length and the signer's identity ID. The verification procedure checks that the group elements and exponents satisfy a specific equation (see (3.1) below), and the standard signature is valid for the tuple (ID, l), where l is the length of the log file.

Our first attack on LogFAS exploits a severe weakness in LogFAS' verification equation, and allows for virtually arbitrary forgeries of log files. Given the signer's public key pk and a valid signature σ for a log file L, an attacker can easily compute a valid signature σ' for a modified log file L'. The forged signature σ' may contain different group elements, but does not change the signature on the log file length. Hence, this attack can not change the length of the log file, while the *content* of the log file can be changed arbitrarily.

The second attack on LogFAS exploits the fact that the signer's identity is not cryptographically bound to the signed log file, but only to the log file length. An attacker might exploit this weakness to deceive a verifier into believing a log file L was signed by some signer ID while it was actually created by a distinct signer $\mathrm{ID}' \neq \mathrm{ID}$, by exchanging the signature on the tuple (ID, l) by a signature for (ID', l). This specific attack is outside of the security model considered in [YPR12b].

[1] This model is slightly reminiscent of identity based cryptography [Sha85].

It is possible to use both of our attacks in combination. Combined, these attacks enable an attacker to forge a valid signature for any log file L' of length l, purportedly signed by some signer ID, iff the attacker has the signer's public key $\mathsf{pk}_{\mathrm{ID}}$ and a valid signature created by ID for a log file L of length l.

BM- and AR-FssAgg. The BM- and AR-FssAgg schemes of [Ma08] are sequentially aggregatable variants of the forward-secure signature schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00]. Both of the original schemes are quite similar, since the scheme by [AR00] is based on [BM99]. In order to support aggregation, the FssAgg constructions use a "de-randomized" signing algorithm, where the independently chosen, random per-signature values r that were used in the original schemes are replaced by arithmetically related values. (New values r_{i+1} are obtained by squaring the previous value r_i in $\mathbb{Z}_N^*$ one or more times, where N is a Blum integer, i.e. a product of two prime numbers p, q with $p \equiv q \equiv 3 \pmod 4$.)

Our attacks exploit this "de-randomization" by building a system of equations from obtained signatures. The core idea of our attack allows us to reduce the number of variables in this system of equations (by using the relation between the r_i) such that the number of variables is not larger than the number of equations. Afterwards, the system of equations can be solved by doing linear algebra "in the exponent". As a result, we obtain the secret key sk_t for some epoch t, where t is reasonably small.

Since our attack crucially relies on the de-randomization, our attacks do not carry over to the original schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00].

The original publication of the attacks [Har17] described the attacks on BM-FssAgg and AR-FssAgg as two distinct attacks. Since then, the author has become aware that the original attack on AR-FssAgg is a special case of a more general attack on AR-FssAgg, which is very similar to the attack on BM-FssAgg. Moreover, the AR- and BM-FssAgg signature schemes can be generalized and recast in a single framework encompassing both schemes. Therefore, this thesis will present both schemes as special cases of a generalized signature scheme, called *de-randomized chain-based* (DRCB) signature scheme. Our attacks are generalized to a single attack on the DRCB scheme.

The original (non-generalized) attacks have been implemented and empirically evaluated by [Har17]. We have not implemented the generalized attack, but present the results of the empirical evaluation of the original attacks.

Outline. The remainder of this chapter is organized as follows. The construction of LogFAS is reviewed in more detail in Section 3.2.1. Details on the two attacks sketched above are given in Section 3.2.2. We point out how an attacker might use these attacks in practice in Section 3.2.3. We briefly review LogFAS' proof of security in Section 3.2.4, and point out the flaw in the proof.

We give a more detailed description of the BM-FssAgg and AR-FssAgg schemes as well as the DRCB signature scheme in Sections 3.3.1 to 3.3.3. Some additional mathematical prerequisites are introduced in Section 3.3.4. The generalized attack is

detailed in Section 3.3.5. We discuss how the original attack on AR-FssAgg described in [Har17] compares to the generalized attack in Section 3.3.6. Section 3.3.7 discusses how our attacks on BM-FssAgg and AR-FssAgg might be used in practice. We briefly point out the flaws in the security proofs of the FssAgg schemes in Section 3.3.8. Finally, we present the results of the empirical evaluation of the attacks in Section 3.3.9.

We do not present new, "fixed" versions of the attacked schemes. Modifying these schemes in order to prevent the attacks outlined here is out-of-scope for this thesis.

3.2. LogFAS

LogFAS [YPR12b] is a forward-secure and aggregate audit log scheme, which features high computational efficiency and compact public key sizes at the expense of large secret keys and signatures.

Before we describe our attacks, we will briefly introduce LogFAS. The reader is referred to [YPR12b; YR11] for a more detailed presentation.

3.2.1. Description of LogFAS

LogFAS assumes a Key Generation Center (KGC) which generates keys for individual signers. Each signer i has an identity ID_i. Signatures consist of several values, some of which can be aggregated. For the remainder of this section, we employ the convention that variables with two indices are aggregated values of several epochs. For instance, $s_{1,l}$ is the aggregation of the values $s_1, \ldots, s_l$.

LogFAS uses three fundamental building blocks: an ordinary signature scheme $\Sigma = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, the Schnorr signature scheme [Sch90; Sch91], and an incremental hash function IH based on a collision-resistant hash function H which is modelled as a random oracle [BR93]. The latter two building blocks are briefly introduced below.

Schnorr Signatures. The Schnorr Signature Scheme [Sch90; Sch91] is based on the hardness of the discrete logarithm problem in some group $\mathbb{G}$. It uses a prime-order subgroup $\mathbb{G}$ of $\mathbb{Z}_p^*$, where p is large a prime, $\mathbb{G}$'s order q is also a large prime, and q divides $p-1$. Let α be a generator of $\mathbb{G}$. A secret key for Schnorr's scheme is $y \leftarrow \mathbb{Z}_q^*$, the corresponding public key is $Y \coloneqq \alpha^y \pmod p$.

In order to sign a message m, choose $r \leftarrow \mathbb{Z}_q^*$, set $R \coloneqq \alpha^r \pmod p$, compute the hash value $e \coloneqq H(m \,\|\, R)$ and set $s \coloneqq r - ey \pmod q$. The signature is the tuple (R, s). To verify such a signature, recompute the hash value $e \coloneqq H(m \,\|\, R)$ (where R is taken from the signature and m is given as input to the verification algorithm). Then check if $R \stackrel{?}{=} Y^e \alpha^s \pmod p$ and return 1 if and only if this holds.

The Schnorr signature scheme can be shown to be secure based on the hardness of the discrete logarithm problem in $\mathbb{G}$, if H is modelled as a random oracle [BR93]. This concludes our brief recap of the Schnorr signature scheme. LogFAS uses the same group setup as Schnorr's scheme, so let $p, q, \mathbb{G}, \alpha$ be fixed.

Incremental Hash Function. Let T be the total number of supported epochs. The key of LogFAS' incremental hash function IH consists of T factors $z_1, \ldots, z_T$. The hash value of a sequence of $l \in [T]$ messages $(m_1, \ldots, m_l)$ is then given by

$$IH(m_1, \ldots, m_l) \coloneqq \sum_{i=1}^{l} H(m_i) z_i \pmod{q}.$$

The security of this hash function can be shown under subset-sum-style assumptions, see the references in [YPR12b; YR11] for details.

LogFAS' Algorithms. In LogFAS, an individual signer's secret key is derived from a central long-term secret $b \in \mathbb{Z}_q^*$ held by the KGC (which can be compared to a secret key of the Schnorr scheme) and several values chosen uniformly at random. Each signer's secret key includes a set of coefficients $z_1, \ldots, z_T$ (derived from b) that form the key of IH. The exact relations between the values in the secret key, the public key and the signature are reflected in our summary of LogFAS' algorithms below, but our attack can be understood without fully comprehending how these values relate to each other.

The algorithms used by LogFAS are given below.

Key Generation.

The KGC chooses a random value $b \in \mathbb{Z}_q^*$ and generates a key pair $(\widehat{\mathsf{sk}}, \widehat{\mathsf{pk}})$ using the signature scheme Σ. The long term private and public keys are $(b, \widehat{\mathsf{sk}})$ and $(B \coloneqq \alpha^{b^{-1} \pmod{q}}, \widehat{\mathsf{pk}})$, respectively. These values constitute the secret and public key of the KGC, and hence are implicitly shared for all signers.

Next, for each identity ID_i, the KGC generates temporary keys for each epoch $t \in [T]$ based on random values $r_t, a_t, d_t, x_t \leftarrow \mathbb{Z}_q^*$. These values are used to create interdependent variables as follows:

$$\begin{aligned} y_t &\coloneqq a_t - d_t && \pmod{q}, \\ z_t &\coloneqq (a_t - x_t) b && \pmod{q}, \\ M_t &\coloneqq \alpha^{x_t - d_t} && \pmod{p}, \quad \text{and} \\ R_t &\coloneqq \alpha^{r_t} && \pmod{p}. \end{aligned}$$

Finally, the KGC generates "tokens" $\beta_t \coloneqq \mathsf{Sign}(\widehat{\mathsf{sk}}, H(\mathrm{ID}_i \,\|\, t))$ for each signer ID_i and each epoch number t. These serve as witnesses that signer ID_i has created at least t signatures. Let $\mathsf{sk}'_t \coloneqq (r_t, y_t, z_t, M_t, R_t, \beta_t)$ for each $t \in [T]$. The initial secret key of ID_i is $\mathsf{sk}_1 = (\mathsf{sk}'_1, \ldots, \mathsf{sk}'_T)$.

Each key sk'_t can only be used to sign a single message. Hence, the secret key must be updated each time a message has been signed.

Key Update.

A signer updates his key sk_t $(t \in [T-1])$ to the next epoch sk_{t+1} by simply erasing r_t, y_t, M_t, and β_t from sk'_t.

Signature Generation.

A LogFAS signature $\sigma_{1,l}$ consists of aggregate-so-far values $s_{1,l} \in \mathbb{Z}_q$ and $M'_{1,l} \in \mathbb{Z}_p^*$, the most recent token β_l, as well as the random group elements R_t and the elements z_t of IH's key for all $t \in [l]$.[2]

Given an aggregate signature $\sigma_{1,l-1}$ for $(m_1, \ldots, m_{l-1})$, a new entry m_l and the temporary secret key $(r_l, y_l, z_l, M_l, R_l, \beta_l)$ for epoch l, first compute the hash value $e_l \coloneqq H(m_l \,\|\, l \,\|\, z_l \,\|\, R_l)$. Then compute $s_l \coloneqq r_l - e_l y_l \pmod q$ and aggregate this value into $s_{1,l} \coloneqq s_{1,l-1} + s_l \pmod q$. Next, set $M'_l \coloneqq M_l^{e_l} \pmod p$ and aggregate this into $M'_{1,l} \coloneqq M'_{1,l-1} M'_l \pmod p$. The new aggregate signature is

$$\sigma_{1,l} \coloneqq (s_{1,l}, M'_{1,l}, \beta_l, ((R_t, z_t))_{t=1}^{l}).$$

Each time a signature has been computed, the signer updates the secret key to the next epoch.

Verification.

To verify an aggregate signature $\sigma_{1,l} = (s_{1,l}, M'_{1,l}, \beta_l, ((R_t, z_t))_{t=1}^{l})$ over l log entries $(m_1, \ldots, m_l)$, one first checks the validity of the token β_l. If

$$\mathsf{Verify}(\widehat{\mathsf{pk}}, H(\mathrm{ID}_i \,\|\, l), \beta_l) = 0,$$

then output 0 and exit. Otherwise, compute

$$z_{1,l} \coloneqq IH(m_1 \,\|\, 1 \,\|\, z_1 \,\|\, R_1, \ \ldots, \ m_l \,\|\, l \,\|\, z_l \,\|\, R_l),$$

and check if

$$\prod_{t=1}^{l} R_t \stackrel{?}{\equiv} M'_{1,l} \cdot B^{z_{1,l}} \cdot \alpha^{s_{1,l}} \pmod p. \tag{3.1}$$

Accept if the equation holds (output 1 and exit). Otherwise, reject the signature (output 0 and exit).

3.2.2. The Attacks

We report two simple and efficient attacks on LogFAS. The first one allows for virtually arbitrary modification of log entries, but cannot change the log file size. It requires only minimal computation and a single signature. This attack contradicts the claimed security of LogFAS. We analyzed the proof of security in [YR11] and found a flaw, resolving this contradiction.

Our second attack allows an adversary to masquerade a signature as originating from another (valid) signer. This attack is outside the formal security model considered in [YPR12b], and therefore does not contradict the claimed security. It nonetheless presents a serious threat, as it undermines the signature's authenticity.

[2] The original scheme in [YPR12b] includes the value e_t in the signature. This value has been omitted here, as e_t can be recomputed by the verifier.

Signature Forgery.

Our first attack can be used to sign any sequence of log messages $(m_1^*, \ldots, m_l^*)$ $(l \in [T])$, provided the attacker has a valid signature for some other sequence of log messages $(m_1, \ldots, m_l)$ of the same length, and knows the public key pk.

On a high level, our attack exploits the fact that the right hand side of (3.1) can be fully determined $M'_{1,l}$. Since $M'_{1,l}$ is part of the signature, an attacker can simply set $M'_{1,l}$ to a value such that the equation holds. Computing the respective value essentially only requires modular multiplication, exponentiation and inversion, which can be implemented efficiently.

Concretely, let $\sigma_{1,l} = (s_{1,l}, M'_{1,l}, \beta_l, ((R_t, z_t))_{t=1}^l)$ be the signature known to the attacker. At first, the adversary computes

$$R_{1,l} = \prod_{t=1}^{l} R_t \pmod p$$

and

$$z^*_{1,l} = IH(m_1^* \,\|\, 1 \,\|\, z_1 \,\|\, R_1, \;\; \ldots, \;\; m_l^* \,\|\, l \,\|\, z_l \,\|\, R_l).$$

(S)he then sets $M^*_{1,l} \coloneqq R_{1,l} \cdot B^{-z^*_{1,l}} \cdot \alpha^{-s_{1,l}} \pmod p$. The forged signature is

$$\sigma^*_{1,l} = (s_{1,l}, M^*_{1,l}, \beta_l, ((R_t, z_t))_{t=1}^l).$$

It is easy to see that this signature will be accepted by the verification algorithm. Since β_l is taken from the original signature, it is a valid signature for $H(\mathrm{ID}_i \,\|\, l)$ and so $\mathsf{Verify}(\widehat{\mathsf{pk}}, H(\mathrm{ID}_i \,\|\, l), \beta_l)$ will return 1, i.e. the first check of the verification algorithm will succeed. Now, by our setup, we have

$$M^*_{1,l} \cdot B^{z^*_{1,l}} \cdot \alpha^{s_{1,l}} \equiv (R_{1,l} \cdot B^{-z^*_{1,l}} \cdot \alpha^{-s_{1,l}}) \cdot B^{z^*_{1,l}} \cdot \alpha^{s_{1,l}} \equiv R_{1,l} \equiv \prod_{t=1}^{l} R_t \pmod p.$$

Therefore, the verification algorithm will accept the signature, and the attack is successful. Note that the attack works for arbitrary $B, s_{1,l}, z_1, \ldots, z_l$ and $R_1, \ldots, R_l$. For each possible combination of these, the attack computes a value $M^*_{1,l}$ which satisfies the verification equation. As stated above, this simple attack is due to the structure of (3.1), where the right hand side can be fully and directly determined by $M'_{1,l}$.

Sender Confusion.

We now turn to our second attack. If an attacker has two aggregate signatures $\sigma_{1,l}$, $\sigma'_{1,l}$ for two sequences of log messages of the same length l, created by different signers ID_i, ID_j the attacker can just exchange the β_l tokens. The receiver will accept $\sigma_{1,l}$ as a signature from ID_j, even though the messages were really signed by signer ID_i, and vice versa. This attack is due to the fact that the identity ID_i of the signer is only bound to the signatures β_t but not to the other signature components $s_{1,l}$, $M'_{1,l}$, R_t and z_t.

Combination.

It is possible to use both of our attacks in combination. Assume an attacker knows the public key pk and a valid signature σ for a log file L of length l, created by the signer with the identity ID.

The attacker may then create a forged log file L' (of the same length l), and obtain a valid signature for L' as follows. Firstly, the attacker may choose arbitrary $s_{1,l}, R_1, \ldots, R_l$ and $z_1, \ldots, z_l$ and compute a suitable $M^*_{1,l}$ according to our first attack. Secondly, the attacker may reuse the β_l token from σ, as in our second attack. Then $(s_{1,l}, M^*_{1,l}, \beta_l, ((R_j, z_j))^l_{j=1})$ is a valid, but forged signature for the log file L'. This signature will be accepted by the verification algorithm, misleading the recipient of the signature to believe that the signer with identity ID has authenticated L'.

3.2.3. Attack Consequences

We present a scenario that shows how our attacks might be used in a real-world attack. Consider a corporate network, where there are multiple servers $S_1, \ldots, S_n$ ($n \in \mathbb{N}$) offering different services. Each server S_i collects information in its log files, and regularly transfers all new log entries together with a signature to some central logging server L. The logging server L checks the signatures, stores the log data, and might examine it automatically for signs of a security breach using an intrusion detection system (IDS). If a server S_i does not transmit any new log entries to L within a certain amount of time, L raises an alarm (as there might be an attacker suppressing the delivery of log messages to L). Assume that LogFAS is used to sign log entries.

An attacker $\mathcal{A}$ who has broken into a server S_i in the corporate network without raising an alarm might retroactively change the log entries not yet transmitted to L to cover his traces, and then create a new (valid) signature for the modified log file using our first attack. The attacker continues to transmit log entries to L regularly, in order not to raise an alarm, albeit $\mathcal{A}$ replaces log entries that might raise suspicion with ones that appear perfectly innocuous.

Now, assume that the attacker can bring her-/himself into a man-in-the-middle position between some other server S_j and L. (This might be achieved using techniques such as ARP spoofing.) $\mathcal{A}$ may now filter and change log entries sent from S_j to L on-the-fly, while our first attack allows him to create valid signatures. Thus, the attacker may attack S_j without risking detection by the IDS at L.

To illustrate our second attack, suppose that the logging system was fixed to prevent the signature forgery. However, bringing himself into a man-in-the-middle position again, the attacker might still exchange the identities of some servers S_j, S_k included in the signature using our sender confusion attack. $\mathcal{A}$ may then try to compromise S_j, while the IDS raises an alarm regarding an attack on S_k. The attacker can thus misdirect the network administrators' efforts to defend their network, giving $\mathcal{A}$ an advantage, or at least gaining time until the administrators notice the deception.

3.2.4. The Proof of Security

In this section we point out the mistake in LogFAS' proof of security that allowed for the false conclusion of LogFAS being secure. While reading this section, the reader should consider the proof in Section 5 of the technical report [YR11] accompanying the LogFAS paper [YPR12b].

The security proof for LogFAS follows a simple and mostly standard scheme. One assumes an attacker $\mathcal{A}$ that breaks LogFAS, and constructs an attacker $\mathcal{F}$ against the Schnorr signature scheme, using $\mathcal{A}$ as a subroutine. $\mathcal{F}$ first guesses an index w of a message block that $\mathcal{A}$ will modify. $\mathcal{F}$'s challenge public key (for the Schnorr scheme) is then embedded into the temporary key pair for that message, the remaining key pairs are set up honestly.

When the attacker outputs a forgery, the proof considers three cases. The first case deals with attackers that actually create a new message together with a valid signature (as does our attack). The second case deals with truncation attacks and the third case models a hash collision.

The error is located in the first case, where the authors conclude that a forgery for an entirely new message must imply a forgery of a Schnorr-type signature, i.e. that the values R_w, s_w (when properly extracted from the LogFAS signature) must be a valid signature for the message m_w. We can see that this conclusion is false, since our attack does not modify the values R_w, s_w at all, but only replaces the original message with an arbitrary one. Thus, the verification algorithm of the Schnorr scheme will reject the signature with very high probability, while the authors conclude that the signature will be accepted.

3.3. The FssAgg Schemes

This section presents the BM-FssAgg scheme (see Section 3.3.1) and the AR-FssAgg scheme [Ma08] (see Section 3.3.2), as well as our generalization of these schemes, called the *de-randomized chain-based* (DRCB) signature scheme (see Section 3.3.3).

We point out that both the BM-FssAgg scheme and the AR-FssAgg scheme are intended to provide *only one* signature per epoch. Hence, the generalized scheme has the same restriction. For all of these schemes, the respective key must be updated every time a message has been signed.

We then introduce some additional prerequisites for the attack in Section 3.3.4, and describe the generalized attack on the DRCB scheme (Section 3.3.5). Section 3.3.6 describes how our original attack on the AR-FssAgg scheme fits into the framework of the generalized attack on the DRCB scheme. We discuss how an attacker might use our attacks on the BM- and AR-FssAgg schemes in practice in Section 3.3.7. Section 3.3.8 briefly points out the flaws in the security proofs of both schemes. Finally, Section 3.3.9 summarizes the results of our experimental evaluation of the attacks.

3.3.1. Description of the BM-FssAgg Scheme

The BM-FssAgg signature scheme [Ma08] is based on a forward-secure signature scheme by Bellare and Miner [BM99]. Both schemes utilize repeated squaring modulo a Blum integer N. (An integer N is called a *Blum integer* if it is a product of two primes p, q such that $p \equiv q \equiv 3 \pmod 4$.) Again, we first describe the BM-FssAgg scheme before we turn to our attack.

Let T be the number of supported epochs and H a hash function that maps arbitrary bit strings to bit strings of some fixed length $l \in \mathbb{N}$.

Intuitively, the scheme is built on $l+1$ sequences of units modulo N, where in each sequence, each number is obtained by squaring the predecessor. Once the starting points r_0 and $s_{i,0}$ (for $i \in [l]$) have been selected during key generation, the scheme successively computes

$$\begin{aligned} r_{j+1} &\coloneqq r_j^2 \pmod N && \text{for } j \in \{0, \dots, T\} \\ s_{i,j+1} &\coloneqq s_{i,j}^2 \pmod N && \text{for } j \in \{0, \dots, T\} \text{ and } i \in [l]. \end{aligned} \tag{3.2}$$

When r_0 and the $s_{i,0}$ are clear from the context, we may thus naturally refer to r_j and $s_{i,j}$ for $j \in [T+1]$ throughout this section. Observe that these sequences form one-way chains: Given any element $s_{i,j}$ of a chain, it is easy to compute the subsequent elements $s_{i,j'}$ with $j' > j$, but it is unknown how to efficiently compute the previous ones without knowing the factorization of N. (The same holds analogously for the chain of the r_j-s.)

We now describe the BM-FssAgg scheme in more detail. See Figure 3.1 for a depiction of the key evolution process.

Key Generation.

Pick two random, sufficiently large primes p, q, each congruent to 3 modulo 4, and compute $N = pq$. Next, pick $l+1$ random integers $r_0, s_{1,0}, \dots s_{l,0} \leftarrow \mathbb{Z}_N^*$. Compute $y \coloneqq 1/r_{T+1} \pmod N$, and $u_i \coloneqq 1/s_{i,T+1} \pmod N$ for all $i \in [l]$. The public key is then defined as

$$\mathsf{pk} \coloneqq (N, T, u_1, \dots u_l, y),$$

whereas the initial secret key is

$$\mathsf{sk}_1 \coloneqq (N, j = 1, T, s_{1,1}, \dots, s_{l,1}, r_1).$$

Key Update.

In order to update the secret key, simply replace all $r_j, s_{i,j}$ by the respective $r_{j+1}, s_{i,j+1}$ (i.e., square all these values), and increment the epoch counter j.

Signing.

In order to sign a message m_j, first compute the hash value $c \coloneqq H(j, y, m)$. Let $c_1, \dots, c_l \in \{0, 1\}$ be the bits of c. The signature for m is

$$\sigma_j \coloneqq r_j \prod_{i=1}^{l} s_{i,j}^{c_i},$$

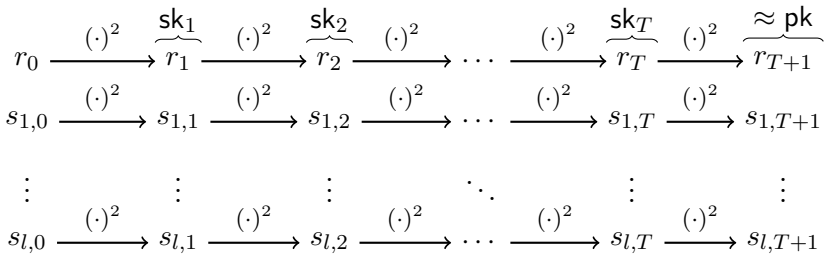

Figure 3.1.: Illustration of the key evolution process of the BM-FssAgg scheme. Each secret key sk_j consists of $l+1$ values $r_j, s_{i,j} \in \mathbb{Z}_N^*$. Each r_{j+1} is computed as r_j^2. The $s_{i,j}$ are computed analogously. The public key contains the modulus N as well as $y \coloneqq 1/r_{T+1} \pmod N$ and all $u_i \coloneqq 1/s_{i,T+1} \pmod N$ for $i \in [l]$.

i.e., the signature is the product of r_j and all $s_{i,j}$ where $c_i = 1$.

An aggregate signature for multiple messages is computed by multiplying the individual signatures. Thus, a signature σ_j can be added to an aggregate signature $\sigma_{1,j-1}$ by computing the new aggregate as

$$\sigma_{1,j} \coloneqq \sigma_{1,j-1} \cdot \sigma_j \pmod N.$$

Verification.

Given an aggregate signature $\sigma_{1,t}$ for messages $m_1, \ldots, m_t$ signed in epochs 1 through t, the verification algorithm will effectively "strip off" the individual signatures one-by-one, starting with the *last* signature.

More precisely, to verify $\sigma_{1,t}$, act as follows: Recompute the hash value $c_t = (c_{1,t}, \ldots, c_{l,t}) \coloneqq H(t, y, m_t)$ of the last message. (Recall that the signature for m_t is $r_t \prod_{i=1}^{l} s_{i,t}^{c_{i,t}}$.) Square $\sigma_{1,t}$ exactly $T+1-t$ times, effectively adding $T+1-t$ to the j-indices of all r_j, $s_{i,j}$ contained in $\sigma_{1,t}$. (In particular, this effectively changes the signature for m_t to $r_{T+1} \prod_{i=1}^{l} s_{i,T+1}^{c_{i,t}}$.) Multiply the result with $y \prod_{i=1}^{l} u_i^{c_{i,t}}$, cancelling out the last signature because y and the u_i are the modular inverses of r_{T+1} and the $s_{i,T+1}$.

For the last-but-one message, square the result another time (projecting the last-but-one signature into the epoch $T+1$), recompute the hash value $c_{t-1} = (c_{1,t-1}, \ldots, c_{l,t-1})$, and cancel out the last-but-one signature by multiplication with $y \prod_{i=1}^{l} u_i^{c_{i,t-1}}$.

The scheme continues analogously for the remaining messages $m_{t-2}, \ldots, m_1$. If the procedure terminates at a value of 1, the aggregate signature is accepted as valid, otherwise it is rejected as invalid.

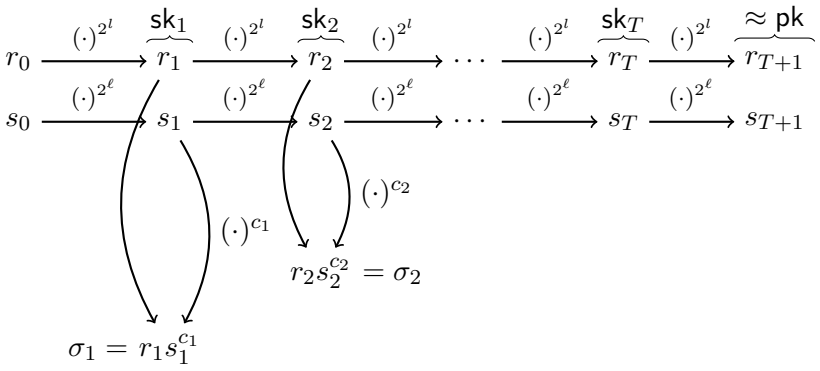

Figure 3.2.: Illustration of the key evolution process and signing process (without aggregation) in AR-FssAgg. The key evolution is very similar to the key evolution in BM-FssAgg, except for the following two differences: Firstly, there is only a single chain of s_j (instead of l chains) in addition to the chain of r_j-s. Secondly, in each chain, each successive element is obtained by raising its predecessor to the power of 2^l (instead of only 2). The signing process combines the values r_j and s_j to $\sigma_j \coloneqq r_j s_j^{c_j}$. Individual signatures can be multiplied to obtain an aggregate signature for multiple messages (not depicted).

3.3.2. Description of the AR-FssAgg Scheme

In the following, we will briefly describe the differences between the AR-FssAgg scheme and the BM-FssAgg scheme. The reader is referred to [Ma08] for a complete description of the AR-FssAgg construction.

The main difference between the AR-FssAgg scheme and the BM-FssAgg scheme is that the former interprets the hash function's output c as an integer in $\{0, \ldots, 2^l - 1\}$, instead of l individual bits. Consequently, the $l+1$ chains of squares $r_j, s_{i,j}$ are replaced by just two chains r_j, s_j of higher powers, namely:

$$\begin{aligned} r_{j+1} &\coloneqq r_j^{(2^l)} \pmod N \qquad \text{for } j \in \{0, \ldots, T\} \\ s_{j+1} &\coloneqq s_j^{(2^l)} \pmod N \qquad \text{for } j \in \{0, \ldots, T\}. \end{aligned}$$

As for the BM-FssAgg scheme, the starting points r_0 and s_0 are chosen randomly, and N is a Blum integer. The key update procedure is adapted canonically: r_j and s_j are raised to their 2^l-th power instead of being squared. Thus, they are replaced by $r_j^{2^l}$ and $s_j^{2^l}$, respectively. In the signing procedure, the hash value c is computed as before, but the signature for the single message is now $\sigma_j \coloneqq r_j \cdot s_j^c \pmod N$. The aggregate signature is $\sigma_{1,j} \coloneqq \sigma_{1,j-1} \cdot \sigma_j \pmod N$, as before.

The process of key evolution as well as signing (without aggregation) is depicted in Figure 3.2.

3.3.3. Generalization

Our description of AR-FssAgg above already pointed out the differences between the BM-FssAgg and the AR-FssAgg scheme. Given this comparison, the generalization of both schemes is straightforward:

The hash output is split into $b \in \mathbb{N}$ blocks of $\mu := l/b$ bits each (where b divides l), and the bits of each block are taken to represent integers $c_i \in \{0, \ldots, 2^\mu - 1\}$. The DRCB scheme uses $b+1$ chains r_j, $s_{i,j}$ ($i \in [b]$). When evolving a secret key, each r_{j+1}, $s_{i,j+1}$ is computed as

$$\begin{array}{rcll} r_{j+1} & := & r_j^{(2^\mu)} & (\text{mod } N) \quad \text{for } j \in \{0, \ldots, T\} \\ s_{i,j+1} & := & s_{i,j}^{(2^\mu)} & (\text{mod } N) \quad \text{for } j \in \{0, \ldots, T\} \text{ and } i \in [b]. \end{array} \tag{3.3}$$

(I.e. each element of the chain is obtained by squaring its predecessor μ times.) The starting values $r_0, s_{i,0}$ are chosen at random during key generation.

The signing algorithm splits the hash output c into b blocks $c_1, \ldots, c_b$ (as described above), and then computes the signature as $\sigma_j := r_j \cdot \prod_{i=1}^{b} s_{i,j}^{c_i}$.

Verification raises the signature $\sigma_{1,t}$ to the power of 2^μ for $T + 1 - t$ times, and multiplies the result with $y \cdot \prod_{i=1}^{b} u_i^{c_i}$ to "strip off" an individual signature, where $y = 1/r_{T+1} \pmod N$ and $u_i = 1/s_{i,T+1} \pmod N$ for $i \in [b]$ are contained in the public key.

Given this generalization, the BM-FssAgg scheme can be obtained by choosing $b = l$, while the AR-FssAgg scheme is obtained by setting $b = 1$.

3.3.4. Attack Prerequisites

Before turning to the attack on the DRCB scheme, we would like to introduce some additional mathematical prerequisites. This enables us to give a more concise presentation of our attack in the next section.

Lemma 3.1 (Repeated Raising to Powers)**.** Let $N, \mu \in \mathbb{N}$, and $f : \mathbb{Z}_N \to \mathbb{Z}_N, x \mapsto x^{(2^\mu)}$. Let f^k denote the k-times iterated repetition of f, i.e. $f^0 = \mathsf{ID}$ and $f^{k+1} : x \mapsto f(f^k(x))$ for $k \in \mathbb{N}_0$, where ID denotes the identity function. Then

$$f^k(x) = x^{(2^{k\mu})} \quad \text{for all } x \in \mathbb{Z}_N, k \in \mathbb{N}_0.$$

Proof. The proof easily follows by induction over k using standard rules for exponentiation. For the start of the induction, let $k = 0, x \in \mathbb{Z}_N$. Then

$$x^{(2^{k\mu})} = x^{(2^0)} = x^1 = x = \mathsf{ID}(x) = f^0(x) = f^k(x),$$

as claimed. For the induction step, assume that $f^k(x) = x^{(2^{k\mu})}$ for a specific $k \in \mathbb{N}_0$ and all $x \in \mathbb{Z}_N$. Then

$$f^{k+1}(x) = f(f^k(x)) = f(x^{(2^{k\mu})}) = \left(x^{(2^{k\mu})}\right)^{(2^\mu)} = x^{(2^{k\mu})(2^\mu)} = x^{(2^{(k+1)\mu})}. \qquad \square$$

Linear Algebra over $\mathbb{Z}$. We now briefly introduce linear algebra over $\mathbb{Z}$. We assume the reader is familiar with basic linear algebra, in particular vector spaces over $\mathbb{R}$, and focus on a few notable differences between linear algebra over $\mathbb{R}$ and $\mathbb{Z}$.

For our attack, we will need to "solve" a matrix over $\mathbb{Z}$. More specifically, given a set of row vectors $c_j \in \mathbb{Z}^d$ ($d \in \mathbb{N}$), we will want to represent the standard basis vectors e_i as integer linear combinations of the c_j.

In the case of vector spaces over $\mathbb{R}$, this problem is easily solved by algorithms like Gaussian elimination. However, over $\mathbb{Z}$, a linearly independent set S of d vectors does not necessarily form a basis of $\mathbb{Z}^d$. Consider, for example, $S = \{(2,0),(0,2)\}$. Clearly, every integer linear combination of these vectors will only have even entries, and there is no way that an integer linear combination of these vectors can yield a vector with an odd component, i.e. a standard basis vector.

Formally, for some dimension $d \in \mathbb{N}$, $\mathbb{Z}^d$ is not a vector space over $\mathbb{Z}$, since $\mathbb{Z}$ is not a field but only a ring. $\mathbb{Z}^d$ is referred to as a *free module* over $\mathbb{Z}$. We will nonetheless continue to refer to elements of $\mathbb{Z}^d$ as "vectors" for simplicity.

For similar reasons, the Gaussian elimination method is *not* suited for "solving" a linear system of equations over $\mathbb{Z}$, since it will compute a linear combination of the vectors if one exists, but the output may not be an *integer* linear combination. We therefore need to employ different algorithms.

Specifically, we compute the *Hermite Normal Form* (HNF) of the matrix to be solved. The exact definitions and conventions used for the HNF differ in the literature. The following definition is a special case of Definition 2.8 given by Adkins and Weintraub [AW92, p. 301], applying the preceding Example 2.7 (1) on the same page, and including an erratum published online [Wei].

Definition 3.2 (Hermite Normal Form)**.** Let $A \in \mathbb{Z}^{m \times n}$ be an integer matrix. Denote the i-th row of A by a_i, and the j-th entry of the i-th row by $a_{i,j}$ (for $i \in [m]$ and $j \in [n]$). A is in *Hermite Normal Form* iff there is a non-negative integer r with $0 \leq r \leq m$ such that

1. $a_i \neq 0$ for all $1 \leq i \leq r$ and $a_i = 0$ for all $r+1 \leq i \leq m$, and
2. there is a sequence of column indices $1 \leq n_1 < \ldots < n_r \leq n$ such that for all $i \in [r]$ the following three conditions hold:

$$\begin{aligned} a_{i,n_i} &> 0 \\ a_{i,j} &= 0 && \text{for} \quad j < n_i, \text{ and} \\ 0 \leq a_{j,n_i} &< a_{i,n_i} && \text{for} \quad 1 \leq j < i. \end{aligned}$$

Intuitively, a matrix is in HNF if only the first r rows are occupied (and the remaining $m-r$ rows are zero), each non-zero row has a positive "pivot" element a_{i,n_i} (which is the first non-zero element in this row), the pivot element of each row is further to the right than the pivot of the preceding row, and all elements *above* a pivot element are between 0 (inclusive) and the pivot (exclusive). See [AW92, Table 2.1, p. 301] for a schematic depiction of matrices in HNF.

We make use of the following two theorems:

Theorem 3.3 (Transformation to HNF)**.** Let $A \in \mathbb{Z}^{m\times n}$. Then there are matrices $H \in \mathbb{Z}^{m\times n}, R \in \mathbb{Z}^{m\times m}$ such that H is in Hermite Normal Form and $H = RA$.

The theorem as noted here is a special case of [AW92, Theorem 2.9, p. 302], again applying [AW92, Example 2.7 (1), p. 301].[3]

Theorem 3.4 (Uniqueness of the HNF)**.** Let $A \in \mathbb{Z}^{m\times n}$. Then the Hermite Normal Form of A is unique.

Again, the theorem is based on [AW92, Theorem 2.13, p. 304], applying [AW92, Example 2.7 (1)]. Furthermore, the HNF is known to be computable in polynomial time, see e.g. [KB79; MW01].

Finally, we will need the following lemma:

Lemma 3.5 (HNF of Bases)**.** Let $A \in \mathbb{Z}^{m\times n}$ be an integer matrix $(m, n \in \mathbb{Z}, m \geq n)$, and let a_i denote the rows of A (for $i \in [m]$). Then the a_i span $\mathbb{Z}^n$ (wrt. integer linear combinations) iff the HNF H of A is of the form

$$H = \begin{pmatrix} \mathbf{1}_n \\ \mathbf{0}_{m-n,n} \end{pmatrix},$$

where $\mathbf{1}_n$ is the $n \times n$ identity matrix and $\mathbf{0}_{m-n,n}$ is the all-zero matrix with $m - n$ rows and n columns.

Proof. Assume that all row vectors $x \in \mathbb{Z}^n$ can be represented as integer linear combinations of the a_i. Then, in particular, the standard basis vectors $e_i = (e_{i,1}, \ldots, e_{i,n}) \in \mathbb{Z}^n$ (where $e_{i,j} = 1$ if $i = j$, and $e_{i,j} = 0$ otherwise) can be represented as integer linear combinations x_i of the rows of A, i.e. $x_i A = e_i$, where $x_i \in \mathbb{Z}^m$. Thus there exists a matrix $R' \in \mathbb{Z}^{n\times m}$ such that $R'A = \mathbf{1}_n$. We build the matrix R by appending $m - n$ zero rows to R'. Then $RA = H$ has the form depicted in the theorem statement. Now, since $RA = H$, and H is in HNF, and the HNF is unique, H must be the HNF of A.

Conversely, if the HNF of A has the form H as described in the theorem statement, then each $v \in \mathbb{Z}^n$ can clearly be represented as an integer linear combination $x \in \mathbb{Z}^m$ of the rows of H, i.e. $v = xH$. Since there exists a matrix R such that $RA = H$, we may as well write $v = x(RA) = (xR)A$, and thus xR defines an integer linear combination of the rows of A that yields v. Since v was chosen arbitrarily, we have shown that the rows of A span $\mathbb{Z}^n$. □

3.3.5. Attack on the Generalized Scheme

Recall that the DRCB scheme divides the hash values of messages into b blocks of μ bits each. We show a conceptually simple way to recover the secret key sk_t $(t \geq b + 1)$ from t successive aggregate signatures and the public key pk. Our attack may work with as little $t = b + 1$ signatures, but has a higher success probability if $t > b + 1$. In our experiments, $t = b + 11$ signatures have been sufficient for all cases.

[3] Cf. [AW92, Remark 2.13 (2), pp. 61–62], [AW92, Example 5.17 (1), p. 86], [AW92, Definitions 1.8 and 1.9, pp. 187–188], and [AW92, Definition 2.1, p. 296].

Attack Overview. Before proceeding to the mathematical details, the author would like to give an informal, high-level overview of the attack. Our attack makes use of the fact that the r_j values, which are supposed to randomize the signatures, are not chosen independently at random, but are strongly interdependent.[4] This allows us to set up a set of equations with a limited number of variables (namely, r_t and the $s_{i,t}$), and then solve the equations for these variables, which together make up the secret key sk_t.

The attack is divided into three main steps. The first step is to recover signatures for single messages from successive aggregate signatures. This step is quite simple, since each aggregate signature $\sigma_{1,j}$ is the product of the individual signatures σ_j up to index j, and we can simply divide (modulo N) $\sigma_{1,j}$ by $\sigma_{1,j-1}$ to recover σ_j.

The second step of our attack is perhaps the most crucial one. Here, we raise each individual signature σ_j to its 2^μ-th power for $t-j$ times, effectively adding $t-j$ to the j-indices of the $r_j, s_{i.j}$ because of (3.3). This process is illustrated in Figure 3.3 for the example of AR-FssAgg. We thus obtain a system of $t \geq b+1$ equations in the $b+1$ variables $r_t, s_{i,t}$ $(i \in [b])$.

The third and final step is to solve this system of equations. Doing so requires us to systematically compute r_t and the $s_{i,t}$ from a system of equations where the $s_{i,t}$ have essentially random exponents. We use linear algebra "in the exponent" to find row operations that reduce the exponents of each equation (considered as a row vector) to the standard basis vectors. An equation having a standard basis vector as exponents directly reveals the corresponding value $s_{i,j}$.

We will now describe our attack in more detail. Fix arbitrary messages $m_1, \ldots, m_t$ and the respective aggregate signatures $\sigma_{1,j}$, where

$$\sigma_{1,j} = \prod_{k=1}^{j} \sigma_k \quad \text{for all } j \in [t].$$

Each aggregate signature $\sigma_{1,j}$ is valid for messages $m_1, \ldots, m_j$. Let $c_{i,j}$ denote the i-th block of the hash value of message m_j, as computed by the signing algorithm.

First Step: Recovering Individual Signatures. Firstly, recover the individual signatures $\sigma_j \coloneqq \sigma_{1,j}/\sigma_{1,j-1} \pmod N$ for all $j \in [t]$, letting $\sigma_{1,0} = 1$. Observe that

$$\begin{array}{ccccccc} \sigma_1 & = & r_1 & s_{1,1}^{c_{1,1}} & \ldots & s_{b,1}^{c_{b,1}} \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ \sigma_t & = & r_t & s_{1,t}^{c_{1,t}} & \ldots & s_{b,t}^{c_{b,t}}. \end{array}$$

For ease of presentation, we let $s_{0,j} = r_j$ and $c_{0,j} = 1$ for all j. We thus have

$$\sigma_j = \prod_{i=0}^{b} s_{i,j}^{c_{i,j}}.$$

[4] For this reason, our attack does not carry over to the underlying forward-secure signature schemes by Bellare and Miner [BM99] and Abdalla and Reyzin [AR00]. In their schemes, the values r_j are chosen uniformly and independently at random, which prevents our attack.

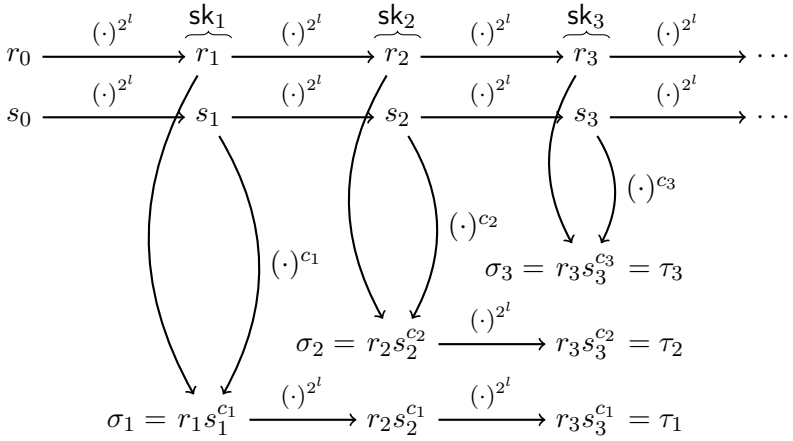

Figure 3.3.: Illustration of the second step of our attack for AR-FssAgg with $t = 3$. By raising each signature σ_j to the power of 2^l for $t - j$ times, we obtain three equations in r_3 and s_3 with known results τ_1, τ_2, τ_3. Note that in the generalized attack, we raise the values to their 2^μ-th power, but $\mu = l$ for AR-FssAgg.

Second Step: "Projection" into the Target Epoch. Secondly, for all $j \in [t]$, we compute:

$$
\begin{aligned}
\tau_j &\coloneqq \sigma_j^{(2^{(t-j)\mu})} \\
&= \left(\prod_{i=0}^{b} s_{i,j}^{c_{i,j}} \right)^{(2^{(t-j)\mu})} \\
&= \prod_{i=0}^{b} \left(s_{i,j}^{c_{i,j}} \right)^{(2^{(t-j)\mu})} \\
&= \prod_{i=0}^{b} \left(s_{i,j}^{(2^{(t-j)\mu})} \right)^{c_{i,j}} \\
&= \prod_{i=0}^{b} s_{i,j+t-j}^{c_{i,j}} && \text{by Lemma 3.1 and (3.3)} \\
&= \prod_{i=0}^{b} s_{i,t}^{c_{i,j}} && (3.4)
\end{aligned}
$$

This process is illustrated by Figure 3.3 for AR-FssAgg with $t = 3$. We thus obtain

$$\begin{array}{ccccccccc} \tau_1 & = & \sigma_1^{(2^{(t-1)\mu})} & = & s_{0,t}^{c_{0,1}} & s_{1,t}^{c_{1,1}} & \cdots & s_{b,t}^{c_{b,1}} \\ \tau_2 & = & \sigma_2^{(2^{(t-2)\mu})} & = & s_{0,t}^{c_{0,2}} & s_{1,t}^{c_{1,2}} & \cdots & s_{b,t}^{c_{b,2}} \\ \vdots & & \vdots & & \vdots & \vdots & \ddots & \vdots \\ \tau_{t-1} & = & \sigma_{t-1}^{(2^{1\mu})} & = & s_{0,t}^{c_{0,t-1}} & s_{1,t}^{c_{1,t-1}} & \cdots & s_{b,t}^{c_{b,t-1}} \\ \tau_t & = & \sigma_t^{(2^{0\mu})} & = & s_{0,t}^{c_{0,t}} & s_{1,t}^{c_{1,t}} & \cdots & s_{b,t}^{c_{b,t}}, \end{array} \tag{3.5}$$

where all $c_{i,j} \in \{0, \ldots, 2^\mu - 1\}$. We thus have $t \geq b+1$ equations in the $b+1$ unknown variables $s_{i,t}$.

Third Step: Solving for $s_{i,t}$. For our third step, we now want to solve these equations for the $s_{i,t}$. Observe that if we compute $\tau_i \cdot \tau_j$ for $i, j \in [t]$, the result is

$$\begin{aligned} \tau_i \cdot \tau_j &= (s_{0,t}^{c_{0,i}} \; s_{1,t}^{c_{1,i}} \; \cdots \; s_{b,t}^{c_{b,i}}) \cdot (s_{0,t}^{c_{0,j}} \; s_{1,t}^{c_{1,j}} \; \cdots \; s_{b,t}^{c_{b,j}}) \\ &= s_{0,t}^{c_{0,i}+c_{0,j}} \; s_{1,t}^{c_{1,i}+c_{1,j}} \; \cdots \; s_{b,t}^{c_{b,i}+c_{b,j}}. \end{aligned}$$

Thus, we have effectively performed an addition of the row vectors $c_i = (c_{0,i}, \ldots c_{b,i})$, $c_j = (c_{0,j}, \ldots, c_{b,j})$ *in the exponent*. We can similarly realize subtraction of row vectors by division (mod N) of the respective τ_i, τ_j, and multiplication of a row vector with a scalar $a \in \mathbb{Z}$ by computing $\tau_i^a \pmod N$.

We may thus use standard algorithms to systematically compute a set of row operations that transforms the row vectors c_j into the standard basis vectors. Having a standard basis vector e_i (with a 1 in position i and zeroes in all other positions) as the exponent directly reveals a single $s_{i,t}$.

More precisely, we consider the $c_{i,j}$ as a matrix C over the integers, and try to express each standard basis vector e_i as an integer linear combination of the row vectors $c_j = (c_{0,j}, \ldots, c_{b,j})$.[5]

Assume that the rows of C span $\mathbb{Z}^{b+1}$. (We will show that this is a realistic assumption given enough signatures in Section 3.3.9). If this is the case, then the Hermite Normal Form of C is

$$H = \begin{pmatrix} \mathbf{1}_{b+1} \\ \mathbf{0}_{t-(b+1),b+1} \end{pmatrix}, \tag{3.6}$$

as shown in Lemma 3.5. In the following, let $e_i = (e_{i,0}, \ldots, e_{i,b}) \in \mathbb{Z}^{b+1}$ be the i-th standard basis vector.

[5] The preceding steps of our attack induce two deviations from common matrix notation: Firstly, the first index i specifies a *column*, while the second index j identifies a *row* (instead of the other way around). Secondly, columns are indexed starting with $i = 0$, while rows start at index $j = 1$. Both of these deviations could be fixed by re-indexing the matrix C, i.e. defining a matrix D with $d_{i,j} = c_{j,i-1}$ and continuing to work with D. However, the author believes that the exposition of the remainder of the attack is easier to understand if we remain with the current notation.

Continuing our attack, we compute the matrix $R = (r_{i,j}) \in \mathbb{Z}^{t \times t}$ that transforms C into its HNF H (i.e., $RC = H$). We then fix $i \in \{0, \ldots, b\}$ and compute

$$\begin{aligned}\prod_{j=1}^{t} (\tau_j)^{r_{i,j}} &= (s_{0,t}^{c_{0,1}} \ldots s_{b,t}^{c_{b,1}})^{r_{i,1}} \cdot \ldots \cdot (s_{0,t}^{c_{0,t}} \ldots s_{b,t}^{c_{b,t}})^{r_{i,t}} \\ &= s_{0,t}^{r_{i,1}c_{0,1}+\ldots+r_{i,t}c_{0,t}} \cdot \ldots \cdot s_{b,t}^{r_{i,1}c_{b,1}+\ldots+r_{i,t}c_{b,t}} \\ &= s_{0,t}^{e_{i,0}} \cdot \ldots \cdot s_{b,t}^{e_{i,b}} \\ &= s_{i,t}\end{aligned}$$

where the first equality follows from substituting the τ_j according to (3.5) and writing out the product, and the second equality can be obtained by sorting the product by the base terms. To see the third and fourth equality, note that the exponents for the $s_{i,t}$ match the i-th row of the matrix $RC = H$, and that the first $b+1$ rows of H are the standard basis vectors (see (3.6)).

Overall, this gives away $s_{i,t}$. Repeating this step for all $i \in \{0, \ldots, b\}$ allows us to reconstruct all $s_{i,t}$, thus leaking the entire secret key sk_t of the t-th epoch. This concludes the description of our attack against the DRCB scheme. In order to obtain an attack against the BM-FssAgg, and AR-FssAgg, one simply instantiates this attack with $b = l$ or $b = 1$, respectively.

Further Generalizations of Our Attack. Finally, we note that our attack can be generalized further to work with t *pairs* of successive aggregate signatures $(\sigma_{1,j_1-1}, \sigma_{1,j_1})$, $\ldots, (\sigma_{1,j_t-1}, \sigma_{1,j_t})$ instead of a single sequence of t successive aggregate signatures $\sigma_{1,1}, \ldots, \sigma_{1,t}$. The latter scenario simply directly yields the t pairs of successive aggregate signatures required by the former one (together with $\sigma_{1,0} = 1$). In the generalized scenario, the attack recovers the secret key sk_{j_t} (instead of sk_t). The second step has to be adjusted to raise all individual signatures σ_j to suitable powers such the τ_j are composed of r_{j_t} and s_{i,j_t} $(i \in [b])$.

3.3.6. The Original Attack on AR-FssAgg

This section briefly points out how the attack on AR-FssAgg originally described in [Har17] differs from the attack on the DRCB scheme described here, and why the original attack is a special case of the generalized attack on the DRCB scheme.

The original attack in [Har17] only differs in the third step. Given several $\tau_j = r_t s_t^{c_j}$, the author's original publication proposed to compute

$$\phi_j \coloneqq \tau_j / \tau_1 = s_t^{c_j - c_1} \qquad \text{for } j \in \{2, \ldots, t\}$$

in order to cancel out r_t. The attack would then continue to compute the greatest common divisor d of all $c_j - c_1$, and (using the extended euclidean algorithm) compute coefficients f_j such that

$$f_2(c_2 - c_1) + \cdots + f_t(c_t - c_1) = d.$$

These are used to calculate

$$\phi_2^{f_2} \cdots \phi_t^{f_t} = s_t^{f_2(c_2-c_1)} \cdots s_t^{f_t(c_t-c_1)} = s_t^d.$$

Given enough signatures, the greatest common divisor of the $c'_j := c_j - c_1$ would eventually be 1, and this would reveal s_t. Finally, it would be easy to recover r_t as $r_t = \tau_1 / s_t^{c_1}$.

However, this procedure essentially realizes a simple algorithm for computing the HNF of a matrix of the following form:

$$\begin{pmatrix} 1 & c_1 \\ 1 & c_2 \\ \vdots & \vdots \\ 1 & c_t \end{pmatrix}$$

Here, the computation of the ϕ_j implements a subtraction of row vectors in the exponent, effectively clearing the first column, such that all but one row vector have a zero in their first entry. The computation of the greatest common divisor and the coefficients f_j determines a way of producing a minimal value of d in the second column. This step is applied "in the exponent" when the ϕ_j are raised to the respective exponents f_j. If $d = 1$, the only remaining step to produce the HNF would be to cancel out non-zero entries in the second column. The attack described in [Har17] skips this step, since it is not necessary for the attack to be successful.

3.3.7. Attack Consequences

Reconsider the scenario from Section 3.2.3, but assume that log entries are signed with the BM-FssAgg or AR-FssAgg scheme instead of LogFAS.

Assume again that an attacker has managed to break into a server S_i without raising an alarm. He may then bring himself into a man-in-the-middle position between another server S_j and L again, and first passively observes several transmissions of log entries from S_j to L, storing the respective signatures.

If at least t signatures for individual messages can be recovered from the (aggregate) signatures sent to L, the attacker can launch the attack described above to recover a recent secret key. He may then attack the server S_j, filtering the log messages sent from S_j to L on-the-fly, and create valid signatures using the known secret key.

While it may seem unnatural that the aggregate signatures observed by the attacker are directly consecutive, it is actually a plausible scenario. For example, this might happen when the server S_j is mostly idle, e.g. at night.

3.3.8. The Proofs of Security

This section briefly points out the flaw in the security proofs of the BM-FssAgg and the AR-FssAgg schemes. These proofs are given in the appendix of [Ma08].

Both proofs give a reduction to the hardness of factoring a Blum integer, assuming an efficient forger $\mathcal{A}$ on the respective scheme, and constructing an attacker $\mathcal{B}$ on the

factorization of N. The proofs are incorrect for they assume that not only $\mathcal{A}$ may use a signing oracle, but $\mathcal{B}$ has access to a signing oracle, too.

3.3.9. Experimental Results

We implemented the original (non-generalized) attacks on the BM-FssAgg and AR-FssAgg schemes in order to verify the attacks, and to empirically determine the number t of signatures required. (Recall that the attacks assume that the matrix C spans $\mathbb{Z}^{l+1}$.) We measured the run times of the attacks, and found that the attacks are entirely practical.

Since the attacks require a number of signatures, we also implemented the key generation, key updating and signing procedures of the two schemes.[6] The implementations are written for the computer algebra system Sage [Ste] and are available at [Har19b].

Our attack implementations miss a number of quite obvious optimizations: We did not parellelize independent tasks, and some computations are repeated during the attacks. Furthermore, since code for Sage is interpreted (instead of compiled to machine language code), we expect that our attacks could be executed significantly faster if implemented in a compiled language. Our measurements should therefore not be regarded as a precise estimate of the resources required for the respective attacks, but as an upper bound.

Experiment Setup.

All experiments used a modulus size of 2048 bit and were conducted on a desktop office PC, equipped with a four-core AMD A10-7850K Radeon R7 processor with a per-core adaptively controlled clock frequency of up to 3.7 GHz, different L1-caches with a total capacity of 256 KiB, two 2 MiB L2-Caches, each shared between two cores, and 14.6 GiB of RAM. The PC was running version 16.04 of the Ubuntu Desktop GNU/Linux operating system, Sage in version 6.7, and Python 2.7.

For our attack on the BM-FssAgg scheme, we used the SHA-224, SHA-256, SHA-384 and SHA-512 hash functions to examine the influence of the hash length l on the runtime of our attack. The BM-FssAgg scheme was instantiated with 512 epochs for the SHA-224, SHA-256 and SHA-384 hash functions, and with 1024 epochs for the SHA-512 hash function. (Recall that the scheme signs exactly one message per epoch, and our attack on the BM-FssAgg scheme requires at least l signatures, where l is the hash length.)[7]

Our implementation of the attacks first collects the minimum required number of signatures and then checks if the respective requirement on the hash values is fulfilled (i.e. the vectors c_j span $\mathbb{Z}^{b+1}$ or the gcd of the c'_j is 1). Once this is the case, the

[6] Our implementation of the schemes is *only* intended to provide a background for our attacks. We did therefore not attempt to harden our implementation against different types of attacks at all.

[7] The number of supported epochs T may be unrealistically low. But since T does not influence the time required for executing our attacks, a small T is sufficient for our demonstration.

attacks are continued as described above. Otherwise, our implementation gradually requests additional signatures until the requirement is met.

For the attack on the BM-FssAgg scheme, the implementation uses $l + 1$ as the minimum number of required signatures, where l is the output length of the hash function. For the original attack on AR-FssAgg, the implementation assumes that the attack always requires at least 3 signatures. This assumption dismisses the theoretically possible, but very improbable event that $c_2 - c_1 = 1$, in which case the attack could be executed with as little as 2 signatures. We have updated this section and Table 3.1 to correctly reflect the theoretical minimum of the number of required signatures.

For both of our schemes, we measured the time that was necessary to collect the total number of signatures. (This includes the time necessary to compute the signatures in the first place, and to update the keys respectively.) For our BM-FssAgg implementation, this time also includes the computation of the Hermite Normal Form of the given matrix, along with the transformation matrix. For the AR-FssAgg attack, the time includes the computation of the gcd of the c'_j, as well as the factors f_j. We refer to these times as the *signature collection times*. The remaining time required for the attacks is referred to as *reconstruction time*. A *measurement* corresponds to one execution of an attack.

Our experiments quickly showed that the reconstruction times for BM-FssAgg were quite long. Given the large amount of time required for the reconstruction and the small amount of variation in the reconstruction times, we restricted our examination of the reconstruction times of BM-FssAgg to 50 measurements per hash-function. For the reconstruction time of the attack on the AR-FssAgg scheme, the number of requested signatures (for both schemes), and the the signature collection times (for both schemes), we collected 250 measurements per scheme and hash-function.

Results

Our results are summarized in Table 3.1. All times are given in seconds.

In our experiments regarding the attack on BM-FssAgg, the greatest difference $d = t - (l + 1)$ between t (the number of actually required signatures) and $l + 1$ (the minimum number of required signatures) was 10. (So, $t = l + 11$ signatures were always sufficient.) For AR-FssAgg, $t = 2 + 5$ signatures have been sufficient for all of our 250 tries. The number of signatures actually required in our experiments is shown in the top third of Table 3.1. The theoretical minimum of signatures required to launch the attacks is given for comparison, denoted as "Theoretical Optimum".

We found that despite the lack of optimizations, our attack on BM-FssAgg took only minutes to recover the respective secret key (in the case of SHA-224) and at most 50 minutes (in the case of SHA-512). Our attack on the AR-FssAgg scheme took less than 0.05 seconds in all 250 measurements.

For BM-FssAgg, the reconstruction time turned out to be the major part of the attack time. In retrospect, this is understandable, since the computation of a single $s_{i,t}$ requires t modular exponentiations, so the reconstruction of all $s_{i,t}$ (including $r_t = s_{0,t}$) required $t \cdot (l + 1) \geq (l + 1)^2$ modular exponentiations.

3.4. Summary

We have presented a total of four attacks on LogFAS [YPR12b], the BM-FssAgg scheme, and the AR-FssAgg scheme [Ma08]. The attacks on LogFAS have been acknowledged by one of LogFAS' authors, and we have demonstrated the practicality of our attacks on BM-FssAgg and AR-FssAgg experimentally. Our attacks allow for virtually arbitrary forgeries, or even reconstruction of the secret key. We conclude that neither of these schemes should be used in practice.

Scheme	BM-FssAgg				AR-FssAgg
Hash Function	SHA-224	SHA-256	SHA-384	SHA-512	SHA-256
Signatures Required					
Theoretical Optimum	225	257	385	513	2
Observed Minimum	226	258	386	514	3
Average	227.15	259.05	387.27	514.97	3.67
Standard Deviation	1.42	1.33	1.73	1.38	0.97
Maximum	234	264	395	522	7
Signature Collection Times					
Minimum	11.74	17.13	65.46	180.02	9.0e-3
Average	22.18	28.79	118.98	292.33	11e-3
Standard Deviation	9.88	12.34	62.34	136.14	3.0e-3
Maximum	67.87	76.59	430.97	1006.61	22e-3
Reconstruction Times					
Minimum	104.06	154.14	580.41	1502.37	6.0e-3
Average	121.48	170.81	634.34	1753.68	9.2e-3
Standard Deviation	9.09	17.17	48.24	126.57	4.4e-3
Maximum	137.94	207.54	736.52	1935.59	24e-3

Table 3.1.: Experimental Results. All times are given in seconds.

4. Secure Logging with Verifiable Excerpts

This chapter presents a scheme for securely storing log files which supports the creation of excerpts of log files. These excerpts can be publicly verified with regard to the authenticity and integrity of the contained log entries as well as with regard to their *completeness*, i.e. the presence of all "relevant" log entries in the excerpt.

This chapter is strongly based on [Har16a; Har16b], and significant parts of the text in this chapter have been taken from these publications without or with only minor changes. Additionally, Sections 4.3 and 4.4.3 reproduce some text from [Har+17b], with modifications.

4.1. Introduction

Log files are generally well-suited to serve as evidence, especially if they can be verified publicly. However, to actually prove a certain fact (e.g. in court) with the help of a log file is problematic *even if* the log file's integrity is unharmed, since the log file may contain confidential information which must not be disclosed. Furthermore, a large fraction of log entries may be irrelevant. Filtering these out significantly facilitates the log file analysis.

We propose a logging scheme that can support the *verification of excerpts* from a log file. Creating an excerpt naturally solves both problems: Log entries that contain irrelevant and possibly confidential information can simply be omitted from the excerpt. Excerpts created with our scheme remain verifiable, and therefore retain their probative force. We illustrate the use of excerpts with two examples.

Example 4.1 (Banking)**.** Consider a bank B which provides financial services to its customers. In order to prove correct behaviour of its computer systems, the bank maintains log files on all transactions on customers' accounts.

When a customer A accuses the bank of fraud or incorrect operation, the bank will want to use its log files to disprove A's allegations. However, submitting the entire log file as evidence to court is not an option, as this would compromise the confidentiality of all transactions recorded, including the ones of other customers. Besides, the log file may also be prohibitively large.

One might alternatively hand the log entries to an independent expert witness, who verifies the log file integrity and then testifies before court on the correct or incorrect operation of the bank. However, this approach eliminates public verifiability, does not solve the problem of the log file size, and still puts the confidentiality of the transactions of all customers at unnecessary risk, even if the expert witness is bound to protect the confidentiality of transactions.

Yet another solution would be to have the entire log file encrypted (under different keys) and to only reveal keys for those log entries that are of interest to the court's proceedings. This would retain the confidentiality of other customers' bank transactions while allowing for public verifiability. But still, this approach does not solve the problem of the log size.

Utilizing a logging scheme with verifiable excerpts, however, the problem at hand is simple: The bank B generates an excerpt from its log files, containing only information on the transactions on A's account and possibly general information, e.g. about the system state. This excerpt is then submitted to court, where it can be verified by the judge and everyone else. If the verification succeeds, the judge may safely consider the information from the excerpt in her/his deliberation.

Example 4.2 (Cloud Auditing)**.** Imagine an organisation O that would like to use the services of a cloud provider, e.g. for storage. O may be legally required to pass regular audits, and must therefore be able to provide documentation of all relevant events in its computer systems. Therefore, the cloud provider C must be able to provide O with verifiable log files, which can then be included in O's audit report.

Now, if C was to hand over all its log files to O, this would reveal details about other customers' usage of C's services, which would most likely violate confidentiality constraints. Furthermore, once again, the entire log files may be too large for transmission by regular means.

Here, as above, audit logging schemes with verifiable excerpts can solve the problem at hand easily. With these, C could simply create an excerpt containing only information that is relevant for O from its log files. This would solve the confidentiality issue while simultaneously lightening the burden induced by the log file's size, while the excerpt can still be checked by the auditors.

Note that simply being able to prove that all log entries contained in the excerpt are authentic is not sufficient for proving that the excerpt speaks the truth: Some relevant information might have been omitted from the excerpt. Thus, when providing excerpts, *completeness* is an essential feature. We informally define *completeness* as the property of containing all "relevant" log entries.

More formally, we assign each log entry to a set of *categories*. Each log entry can belong to any number of categories, and categories may overlap arbitrarily. The set of categories needs not be fixed in advance, but new categories can be added on-the-fly. However, previously added log entries cannot be added to newly introduced categories, since this would constitute a retroactive modification of the log data, which is what we are striving to prevent.

Each category naturally defines a subsequence of all log entries in the log file. An excerpt may consist of one or more categories, and is defined as the "union" of all the category subsequences.

We achieve completeness by numbering log entries in each category separately. This way, when a certain category ν is supposed to be contained in an excerpt, one can verify that the counters for category ν signed together with the log entries form a strictly increasing sequence without gaps.

This chapter also introduces our first cryptographic logging scheme which features security against truncation attacks. Proving this property requires formally modeling truncation resistance in our security notions. The publications underlying this chapter [Har16a; Har16b] were the first publications which gave a security definition capturing this requirement as well as completeness of excerpts.

Moreover, this section introduces a novel technique for achieving truncation security in the standalone model: We require the log signer to a) sign at least one log entry per epoch (e.g. by adding epoch markers), and b) during verification, provide proof that (s)he knows the secret key sk_t for a t that is "close" to the end of the verified log file. This approach is widely applicable and of independent interest. The use of this technique is illustrated by the two logging schemes constructed in this thesis: our scheme for logging with verifiable excerpts given in this chapter, and our scheme for robust logging with sub-linear storage overhead given in Chapter 5. Both schemes enjoy a formal proof of security with respect to our security notions capturing the truncation security property, making them the first cryptographic logging schemes with provable truncation resistance, since the LogFAS scheme was broken in Chapter 3.

Our scheme makes efficient use of a forward-secure signature scheme, which is used in a black-box fashion. Therefore, our scheme can be instantiated with an arbitrary forward-secure signature scheme and thereby tuned to meet specific performance goals, and be based on a wide variety of hardness assumptions. We analyze our scheme formally and give a perfectly tight reduction to the security of the underlying forward-secure signature scheme.

Outline.

Section 4.2 introduces preliminary definitions and some notation. Section 4.3 discusses truncation security in more detail and presents our approach to achieve this security property. In Section 4.4, we develop a formal framework to reason about log files with excerpts, and give a security definition for such schemes. Section 4.5 presents our construction, proves that it satisfies the security notion from Section 4.4, and analyzes the overhead imposed by our scheme. It also compares our scheme to other schemes from the literature. Finally, Section 4.6 concludes this chapter.

4.2. Preliminaries, Notation and Conventions

We briefly introduce some additional notations and definitions required for this chapter. Recall our definition of index sequences and subsequences from Section 2.2: Given a sequence $S = (s_1, \ldots, s_l)$ of length $l \in \mathbb{N}_0$, an index sequence for S is a strictly increasing sequence $I = (i_1, \ldots, i_n)$ $(n \leq l)$ of integers $i \in [l]$. The subsequence of S induced by I is $S[I] := (s_{i_1}, \ldots, s_{i_n})$.

Definition 4.3 (Operations on Subsequences)**.** Let $S = (s_1, \ldots, s_l)$; let $I = (i_1, \ldots, i_v)$, $J = (j_1, \ldots, j_w)$ be two index sequences for S, and let $T = S[I] = (s_{i_1}, \ldots, s_{i_v})$,

$U = S[J] = (s_{j_1}, \ldots, s_{j_w})$ be the subsequences of S induced by I and J, respectively. Then:

$T \cup U$
: is the subsequence of S that contains exactly those elements s_k for which $k \in I$ or $k \in J$ or both, in the order of increasing $k \in [l]$,

$T \cap U$
: is the subsequence of S that contains exactly those elements s_k for which $k \in I$ and $k \in J$, in the order of increasing $k \in [l]$.

Note that if S contains duplicates, then there may be different index sequences inducing the same subsequence. Therefore, the operations from Definition 4.3 are only well-defined if the index sequences I and J are given. In this work, we will omit specifying I and J when they are clear from the context.

Example 4.4. Let $S = (s_1, \ldots, s_6)$, and let $I := (1, 4, 6)$, $J := (3, 4, 5)$ induce the subsequences $T = (s_1, s_4, s_6)$ and $U = (s_3, s_4, s_5)$ of S. Then we have $T \cup U = (s_1, s_3, s_4, s_5, s_6)$ and $T \cap U = (s_4)$. Note that even if, e.g. $s_5 = s_6$, we would still have $T \cap U = (s_4)$, since the operations are defined based in the indices i of the elements s_i in the sequence S, not based on the equality in the domain $D \supset \{s_1, \ldots, s_6\}$.

4.3. Truncation Security

We present a new approach to achieve truncation security in the standalone model. This approach first appeared in [Har16a; Har16b].

Truncation security refers to the property of a logging scheme to detect log truncations, i.e. the deletion of a tail-end subsequence of the entire log file. Phrased differently, if M is the complete log file, the logging system should be able to correctly identify *any prefix* P of M as incomplete. The logging system must achieve this capability in spite of the fact that P represented the complete log file at some point in the past.

Truncation security can be achieved easily if one assumes the presence of external auditors or other external utilities. For example, in Merkle-tree-based schemes such as [CW09; Bul+14; RFC6962] one simply sends the root of the Merkle tree to the auditors in regular intervals. The Merkle-tree structure even allows for efficient proofs that a newly sent root belongs to a tree representing a log that is a "continuation" of the previous log state. However, this thesis deals with secure logging in the standalone model, i.e. we do not assume the presence of external (trusted) entities. Thus, we need to achieve truncation security by purely cryptographical/mathematical techniques.

At first, this problem seems paradoxical, since signatures usually present a solution to a certain mathematical/computational problem (posed by the verification algorithm), and this solution is independent of when it is verified and what happened since the solution was found.

In fact, detecting log truncations in the standalone model is a surprisingly hard problem. It is easy to create logging schemes where newly added signatures implicitly

re-authenticate previously added log entries, e.g. by arranging the log entries in a hash chain. However, it is unclear how to protect the end of such a chain.

Ma and Tsudik [MT09] were the first to present a mechanism to detect truncations of log files. Their solution is based on forward-secure sequential aggregate signatures. The core idea of their solution lies in the fact that for specific (sequential) aggregate signature schemes (such as [Bon+03]), removing a message from a given aggregate signature is intractable under standard assumptions. For example, for the BGLS signature scheme, the aggregate of n signatures $\sigma_1, \ldots, \sigma_n$ is simply the product $\sigma = \sigma_1 \cdots \sigma_n$ of the signatures in a specific algebraic group. Being able to remove l signatures $\sigma_{n-l+1}, \ldots, \sigma_n$ and thus producing $\sigma' = \sigma_1 \cdots \sigma_{n-l}$ given only σ is equivalent to computing the signature $\sigma'' = \sigma_{n-l+1} \cdots \sigma_n$, since (given σ and σ') it is easy to compute $\sigma'' := \sigma/\sigma'$. (Coron and Naccache [CN03] formally prove this problem to be equivalent to the co-CDH assumption.)

We will not go into more detail on this problem here, but refer the reader to [Kai20], where this issue is discussed in more detail. Moreover, Fischlin, Lehmann, and Schröder [FLS12] as well as Saxena, Misra, and Dhar [SMD14] discuss security notions for sequential and standard aggregate signature schemes, respectively, where the security notions capture such removals of signatures.

Ma and Tsudik [MT09] use this property by keeping only a single signature for the entire log file, which is an aggregate of the signatures for each individual message. The hardness of removing signatures from the aggregate is then used to guarantee that no efficient attacker has a non-negligible chance of removing any message from the log file, and successfully forging a signature for the modified log file.

Note that this property does not hold for log entries signed in the break-in epoch: If an attacker obtains some secret key sk_t, the attacker can recompute all the signatures added during epoch t and thus remove them from the aggregate signature by division.[1] Hence, after break-in, an attacker can truncate the log file to the state it had at the time of the most recent epoch switch.

Ma and Tsudik [MT08; MT09] also propose "immutable" schemes which store signatures for individual log entries in addition to an aggregate signature. These schemes, however, only offer protection against attackers that try to truncate the log file to a state before the most recent "anchor point".

We now describe a new approach of obtaining truncation security in the standalone model, as presented in [Har16a]. The approach follows the idea of verifying that the log file is reasonably up-to-date by

- enforcing that the log file contains at least one entry in each epoch, and
- requiring the log signer to "prove" he owns a secret key sk_t for a t that is "close" to the maximum epoch number of all entries in the log file.

In more detail, we design the logging system to use epoch markers, created every time the secret key is updated. (The epoch markers are signed just before updating the

[1] The signatures of the BLGS scheme [Bon+03] are computed deterministically, thus the attacker can recompute exact copies of the signatures.

secret key. Thus, when a secret key sk_t is updated to sk_{t+1}, the epoch marker is signed with sk_t.) This enforces that there is at least one log entry in each epoch, realizing the first requirement of our technique.

For the second requirement, assume that the last log entry in the log file was signed using sk_t. In this case, we force the log signer to produce a valid signature using either sk_t or sk_{t+1}.[2] What message is signed for this proof is a more or less arbitrary choice. The scheme given in Section 4.5 requires the logger to sign the entire excerpt (with some metadata). Our scheme in Section 5.4.2 has the signer precompute a signature on the log file length. One might just as well imagine a standard challenge-response protocol where the verifier chooses a random nonce r, and the signer authenticates r with the current secret key.[3] One might even use zero-knowledge protocols or non-interactive zero-knowledge schemes to conduct this proof, but we do not pursue this line of research further here.

Note that this approach does not prevent an attacker who broke in during epoch t from "embezzling" the log entries added during epoch t: He might truncate the log file to the state of the most recent epoch switch, and then prove ownership of sk_t. Hence, our approach has the same restriction as the solution by [MT09]. While it is possible to bypass this restriction and achieve full truncation security by simply updating the secret key every time a log entry has been added, this change may negatively impact the efficiency of the logging scheme. Creating a logging scheme that achieves truncation-security in the standalone model even for the most recently added log entry without requiring an epoch switch is an interesting and important open problem.

The restriction on truncation security outlined above is reflected in our security notions, e.g. Definitions 4.13 and 5.19. These notions consider an attacker's forgery trivial, iff:

- the attacker obtained a signature for the forgery from its oracles, or
- the attacker requested the secret key during some epoch t_{BreakIn} and the forgery does not differ from the "real" log file regarding the epochs 1 through $t_{\mathsf{BreakIn}} - 1$.

The former condition is analogous to standard definitions of unforgeability, e.g. existential unforgeability under chosen message attacks [GMR84; GMR88], or its forward-secure variant (see [BM99] or Definition 2.20).

The latter condition only applies if the attacker did use the BreakIn oracle to obtain a secret key. If so, it is considered trivial for the attacker to truncate the log file to the most recent epoch switch (from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn}). Moreover, it is trivial to "extend" the log file from there by adding new, forged log records with arbitrary contents, using the logging scheme's key evolution and signing procedures to create valid signatures. Hence, only changes made to the log file before the epoch t_{BreakIn}

[2] If the last signature added to the log file is an epoch marker, it is legitimate for the log signer to only own sk_{t+1} instead of sk_t.

[3] The scheme would have to make sure that the signature for r can not be mistaken for a signature for a log entry, or otherwise a malicious verifier might abuse this challenge-response protocol as a signing oracle for log entries.

are considered non-trivial. This is accordance with our security model as depicted in Figure 1.1 on page 9.

The security notion given in this chapter was one of the first notions to capture security against truncations. In fact, to the best of the author's knowledge, the only prior notion considering truncation security was given by Yavuz, Peng, and Reiter [YPR12b]. However, their security notion allows for almost arbitrary omission of log entries. For example, after querying the signature oracle three times for messages m_1, m_2, m_3, respectively, a forgery for the log file (m_1, m_3) is considered trivial in their notion. While our scheme supports the creation of signatures for *excerpts* (i.e. subsequences) of the entire log file, too, our notion requires that such excerpts are *complete*. In contrast, the security notion given in [YPR12b] does not consider completeness.

4.4. Secure Logging with Verifiable Excerpts

We now develop a formal model for log files with excerpts. Obviously, given a log file M, an excerpt E is a subsequence of M. However, a scheme where *each* subsequence of M can be verified[4] is not sufficient for our applications, since the provider of the excerpt could simply omit some critical log entries. Put differently, such a scheme may guarantee correctness of all log entries in the excerpt, but it does not guarantee that all relevant log entries are present.

To address this problem, we introduce *categories*. Each log entry is assigned to one or more categories, which may overlap. Each category has an unique name $\nu \in \{0,1\}^*$. We require that when a new log entry m is appended to the log file, one must also specify the names of all categories that m is assigned to.

We return to our banking example from Section 4.1 to illustrate the use of such categories. The bank B introduces a category C_A for each customer A, and then adds each log entry concerning A's account to C_A. The problem of checking the completeness of the excerpt for A's account is thereby reduced to checking the presence of all log entries from the category C_A and possibly from other categories containing general information. Of course, categories may also be added based on other criteria, such as the event type (e.g. creation and termination of an account, deposition or withdrawal of funds, and many more). Note that the set of categories is not fixed in advance; rather the bank must be able to add new categories on-the-fly, as it gains new customers. The use of categories is similar in the cloud provider example.

4.4.1. Categorized Logging Schemes

Definition 4.5 (Categorized Messages and Log Files)**.** A *categorized message* (also *categorized log entry* or *categorized log record*) $m = (N, m')$ is a pair of a finite, non-

[4] This capability is offered by a number of schemes such as LogFAS [YPR12b; YR11] and schemes based on Merkle hash trees, e.g. [CW09; Bul+14].

empty set N of category names $\nu \in \{0,1\}^*$ and a log entry $m' \in \{0,1\}^*$.[5] A *categorized log file* $M = (m_1, \ldots, m_l)$ is a finite, possibly empty sequence of categorized log entries m.

When it is clear from the context that we mean categorized log entries or categorized log files, we will omit the term "categorized" for the sake of brevity. In particular, this chapter will mainly be concerned with categorized log entries and categorized log files.

Definition 4.6 (Categories)**.** A *category* with name $\nu \in \{0,1\}^*$ of a categorized log file $M = ((N_i, m_i'))_{i=1}^{l}$ is the (possibly empty) subsequence C of M that contains exactly those log entries $(N_i, m_i') \in M$ where $\nu \in N_i$. C is denoted by $C(\nu, M)$. C's *index sequence* $I(\nu, M)$ is the (possibly empty, strictly increasing) sequence that contains all $i \in [l]$ for which $\nu \in N_i$.

Definition 4.7 (Excerpts)**.** Given a categorized log file $M = (m_i)_{i=1}^{l}$ and a finite set N of category names, the *excerpt for* N is $E(N, M) = \bigcup_{\nu \in N} C(\nu, M)$. The *index sequence* $I(N, M)$ is the (possibly empty, strictly increasing) sequence of all i with $i \in I(\nu, M)$ for at least one $\nu \in N$.

Clearly, $C(\nu, M)$ is induced by $I(\nu, M)$, and $E(N, M)$ is induced by $I(N, M)$. In the following, we will mostly omit the second parameter, since it will be clear from the context.

We employ the convention that there are two designated, special-purpose categories named "`All`" and "`EM`". The category "`EM`" will contain all epoch markers and no other log entries. A log entry $m = (N, m')$ which is an epoch marker has $N = \{$"`All`", "`EM`"$\}$. By convention, the category "`EM`" is contained in all excerpts. The category "`All`" contains all log entries, i.e. "`All`" $\in N_1 \cap \ldots \cap N_l$, and thus $C($"`All`"$) = M$. As a special case of excerpts, we obtain the entire log file M as an excerpt for the categories $N = \{$"`All`", "`EM`"$\}$.[6]

In the following, variables with two indices correspond to a sequence of values ranging from the first to the second index, i.e. $\sigma_{1,j}$ is the sequence of $(\sigma_1, \ldots, \sigma_j)$, and $M_{1,j} := (m_1, \ldots, m_j)$.

Definition 4.8 (Categorized Key-Evolving Audit Log Scheme)**.** A *categorized key-evolving audit log scheme* is a quintuple of probabilistic polynomial time algorithms $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Extract}, \mathsf{Append}, \mathsf{Verify})$, where:

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk}, \sigma_{0,1})$
: outputs an initial signing key sk_1, a permanent verification key pk, and an initial signature $\sigma_{1,0}$ for the empty log file. T is the number of supported epochs.

[5] The set N of category names is represented by the upper case version of the greek letter "ν", which unfortunately looks identical to the upper case latin letter "n".

[6] Since all epoch markers also belong to the category "`All`", giving "`EM`" is actually redundant here. It is nonetheless specified here for consistency with our requirement that the category "`EM`" must be requested in all excerpts.

$\mathsf{Update}(\mathsf{sk}_t, M_{1,i}, \sigma_{1,i}) \to \mathsf{sk}_{t+1}$
: evolves the secret key sk_t for epoch t to the subsequent signing key sk_{t+1} and then outputs sk_{t+1}. sk_t is erased securely. Update may also use and modify the current log file $M_{1,i}$ as well as the current signature $\sigma_{1,i}$, e.g. by adding epoch markers.

$\mathsf{Extract}(\mathsf{sk}_t, M_{1,i}, \sigma_{1,i}, N) \to \sigma$
: takes a log file $M_{1,i}$ together with a signature $\sigma_{1,i}$ for $M_{1,i}$ and a set N of category names and outputs a signature σ for the excerpt $E(N) = E(N, M_{1,i})$, computed with the help of sk_t.

$\mathsf{Append}(\mathsf{sk}_t, M_{1,i-1}, \sigma_{1,i-1}, m_i) \to \sigma_{1,i}$
: takes as input the secret key sk_t, the current log file $M_{1,i-1}$, its signature $\sigma_{1,i-1}$ and a new (categorized) log entry m_i and outputs a signature $\sigma_{1,i}$ for $M_{1,i} \coloneqq M_{1,i-1} \,\|\, m_i$.

$\mathsf{Verify}(\mathsf{pk}, N, E, \sigma) \to 0/1$
: is given the verification key pk, a set $N = \{\nu_1, \ldots, \nu_n\}$ of category names, an excerpt E and a signature σ. It outputs 1 or 0, where 1 means E is an authentic, unmodified and complete excerpt for categories N, and 0 means E is not. As noted above, one can verify the entire log file up until the current epoch by choosing $N = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{EM}\text{”}\}$.

We require correctness as defined below.

Definition 4.9 (Valid and Regular Signatures)**.** Let LS be a quintuple of algorithms as defined above and $\kappa \in \mathbb{N}$. We say that a signature σ is *valid* for (pk, N, E) iff $\mathsf{Verify}(\mathsf{pk}, N, E, \sigma) = 1$. As always, we may omit pk and/or N when they are clear from the context.

We say that a signature σ is *regular* for (pk, N, E), iff $(\mathsf{pk}, N, E, \sigma)$ are in the image of the following process for some $\kappa \in \mathbb{N}$, $T = T(\kappa) \in \mathrm{poly}(\kappa)$, a log file length $l \in \mathbb{N}$, a categorized log file $M_{1,l} = (m_1, \ldots, m_l)$, an increasing sequence $I = (t_1, \ldots, t_l, t_{l+1})$ with $t_i \in [T]$ for all $i \in [l+1]$, and a set of category names N. The process for creating $(\mathsf{pk}, N, E, \sigma)$ is as follows:

1. Let $(\mathsf{sk}_1, \mathsf{pk}, \sigma_{1,0}) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$, $t \coloneqq 1$, $M_{1,0} \coloneqq ()$, and $\sigma \coloneqq \sigma_{1,0}$.
2. Iterate over all $i \in [l]$ in increasing order:
   a) While $t_i > t$, compute $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t, M_{1,i-1}, \sigma)$ and set $t \coloneqq t + 1$. (Recall that Update may modify σ and $M_{1,i-1}$.)
   b) Set $\sigma \coloneqq \mathsf{Append}(\mathsf{sk}_{t_i}, M_{1,i-1}, \sigma, m_i)$.
   c) Set $M_{1,i} \coloneqq M_{1,i-1} \,\|\, m_i$.
3. While $t_{l+1} > t$, compute $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t, M_{1,l}, \sigma)$ and set $t \coloneqq t + 1$. (Again, Update may modify $M_{1,l}$ and σ.)

4. Output pk, N, $E(N, M_{1,l})$, $\sigma := \mathsf{Extract}(\mathsf{sk}_{t_{l+1}}, M_{1,l}, \sigma, N)$.

The process used for the definition of correctness models regular usage of LS. Here, the m_i are the log entries to be added, and each t_i corresponds to the epoch during which m_i is added to the log file. The additional epoch number t_{l+1} models epoch switches performed after the last log entry was added.

Definition 4.10 (Correctness)**.** Let $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Extract}, \mathsf{Append}, \mathsf{Verify})$ be quintuple of algorithms as defined above. We say that LS is *correct* iff for all $\kappa \in \mathbb{N}$, $T := T(\kappa) \in \mathrm{poly}(\kappa)$, all (pk, N, E) and all signatures σ regular for (pk, N, E) we have that σ is valid for (pk, N, E).

Note that we require Verify to validate excerpts without actually "knowing" the complete log file. This is the main difficulty that our construction must overcome.

Moreover, note that our definition of a logging scheme given above includes only a single verification algorithm, which is intended to verify excerpts. We did not formalize another verification algorithm for complete log files. Rather, we treat the verification of entire log files as a special case of verifying excerpts, namely excerpts for the categories $\{\text{“}\texttt{All}\text{”}, \text{“}\texttt{EM}\text{”}\}$.

4.4.2. General Remarks

Remark 4.11 (Reset Attacks)**.** It is quite obvious that once an attacker has seen a valid signature σ for a log file M from some point in time t_1, (s)he can reset the entire log file to M and restore the previous signature σ once (s)he has control over the log server. Since one requires that $\mathsf{Verify}(\mathsf{pk}, \{\text{“}\texttt{All}\text{”}\}, M, \sigma) = 1$ at t_1, we cannot expect $\mathsf{Verify}(\mathsf{pk}, \{\text{“}\texttt{All}\text{”}\}, M, \sigma) = 0$ at some later point in time t_2, unless Verify has an additional trusted input such as the current time or the number of messages that have been added to the log file so far.

We therefore take a different path and let excerpts remain (cryptographically) valid for an indefinite amount of time. It is then up to the application to decide whether an excerpt is "fresh enough". This is sufficient for both our examples, where only an a posteriori verification of events is required, and everyone can see whether an excerpt spans the time period of interest. In more interactive scenarios, one might consider using a challenge response protocol, as discussed in Section 4.3.

Remark 4.12 (Secret Keys for Generation of Excerpts)**.** In our model, creating an excerpt signature from a log file M and a corresponding signature σ requires a secret key. Our reason for requiring the secret key during extraction is that (intuitively) an attacker should be unable to produce a valid signature for an excerpt. If the generation of an excerpt signature was a public operation, then anyone, and in particular the adversary, could create an excerpt, which would violate our intuition.

Note that requiring a secret key for creating excerpts does not prevent attackers who *do* break in and obtain the secret key from computing valid signatures for excerpts.

4.4.3. Security Model

We now define our security notion for categorized key-evolving audit log schemes. The security notion for logging schemes is similar to the FS-EUF-CMA notion for forward-secure signature schemes, but models the real world setting of secure logging more closely: The log server maintains some internal state which the adversary influences only through his oracles. In more detail, a signature oracle appends an entry to the log file, and an adversary can never again add messages to any earlier state of the log file. Moreover, we have added an oracle which can be used by an adversary to create signatures for log file excerpts.

Definition 4.13 (Forward-Secure Existential Unforgeability under Chosen Log Message Attacks)**.** We define the following experiment for a categorized key-evolving audit log scheme $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Extract}, \mathsf{Append}, \mathsf{Verify})$, a PPT adversary $\mathcal{A}$, the security parameter $\kappa \in \mathbb{N}$, and the number of epochs $T := T(\kappa) \in \mathrm{poly}(\kappa)$:

Setup Phase.

The experiment generates the initial secret key, the public key and the initial signature as $(\mathsf{sk}_1, \mathsf{pk}, \sigma_{1,0}) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$. It initializes the epoch counter $t := 1$, the message counter $i := 1$, and the log file $M_{1,0} := ()$. It then starts $\mathcal{A}$ with inputs pk, 1^T and $\sigma_{1,0}$.

Query Phase.

During the query phase, the adversary may adaptively issue queries to the following three oracles:

Signature Oracle.

Whenever $\mathcal{A}$ submits a message $m_i = (N_i, m_i')$ to the signature oracle, the experiment verifies that "`All`" $\in N_i$ and "`EM`" $\notin N_i$. If these requirements are not met, then the experiment outputs 0 and aborts. Otherwise, the experiment appends m_i to the log file by setting $M_{1,i} := M_{1,i-1} \,\|\, m_i$ and updates the signature to

$$\sigma_{1,i} := \mathsf{Append}(\mathsf{sk}_t, M_{1,i-1}, \sigma_{1,i-1}, m_i).$$

It then sets $i := i + 1$. The oracle returns the new signature $\sigma_{1,i}$.

Epoch Switching Oracle.

Upon a query to the $\mathsf{NextEpoch}$ oracle, the experiment moves to the next epoch, updating the secret key to $\mathsf{sk}_{t+1} := \mathsf{Update}(\mathsf{sk}_t, M_{1,i}, \sigma_{1,i})$, which may update the log file and its signature as a side effect, and incrementing the epoch counter $t := t + 1$. The oracle returns the updated log file M' and signature σ' to the attacker. This oracle may be queried at most $T - 1$ times.

Extraction Oracle.

On input of a set N of category names, the experiment checks if "`EM`" $\in N$. If "`EM`" $\in N$, the experiment creates a signature $\sigma := \mathsf{Extract}(\mathsf{sk}_t, M_{1,i}, \sigma_{1,i}, N)$

for the excerpt $E := E(N, M_{1,i})$ and returns (E, σ) to the adversary. Otherwise, the experiment outputs 0 and aborts.

Break-In Phase.

When the adversary is done with the query phase, the experiment enters the break-in phase.

During this phase, the adversary is no longer allowed queries to the previously defined oracles.[7] Instead, the attacker is provided with a $\mathsf{BreakIn}$ oracle, which returns the current secret key sk_t. If $\mathcal{A}$ queries the $\mathsf{BreakIn}$ oracle, the experiment sets $t_{\mathsf{BreakIn}} := t$. Otherwise, let $t_{\mathsf{BreakIn}} := \infty$.

Forgery Phase.

At the end of the experiment, $\mathcal{A}$ outputs a non-empty, finite set N^* of category names, a forged excerpt E^* for N^*, and a forged signature σ^* for E^*.

We say that (N^*, E^*) is trivial, iff:

- $(N^*, E^*) \in \{(N_1, E_1), \ldots, (N_q, E_q)\}$, where q is the number of queries to the extraction oracle $\mathcal{A}$ did, N_i is the set of categories during $\mathcal{A}$'s i-th query to this oracle ($i \in [q]$), and E_i is the excerpt returned during that call, or
- $\mathcal{A}$ queried the $\mathsf{BreakIn}$ oracle during some epoch t_{BreakIn} and $E' = E(N^*, M')$ is a prefix of E^*, where M' is the log file directly after the epoch switch from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn} (including changes made by Update, if any). We let $M' := ()$ if $t_{\mathsf{BreakIn}} = 1$.

The experiment outputs 1 iff σ^* is valid for (pk, N^*, E^*) and (N^*, E^*) is non-trivial. Otherwise, the experiment outputs 0.

We say that $\mathcal{A}$ *wins* the experiment, iff the experiment outputs 1, otherwise $\mathcal{A}$ *loses* the experiment.

A categorized key-evolving audit log scheme LS is said to be *forward-secure existentially unforgeable under chosen log message attacks* (or *FS-EUF-CLMA-secure*), iff for all $T = T(\kappa) \in \mathrm{poly}(\kappa)$ and all probabilistic polynomial time attackers $\mathcal{A}$:

$$\Pr[\mathcal{A} \text{ wins}] \leq \mathsf{negl}(\kappa).$$

Let us review the above definition. As for standard security notions, we let the adversary completely determine the input to the cryptographic scheme, except for the keys. In our case, this input consists of the messages being submitted to the log (using the signature oracle) as well as the *timing* of these messages (controlled by the order in which $\mathcal{A}$ submits these to the signing oracle as well as the $\mathsf{NextEpoch}$ oracle). While such a powerful adversary may be unrealistic in most real-world scenarios, giving the adversary such power in the experiment results in a stronger security notion.[8]

[7] Again, this restriction is without loss of generality, see Remark 2.21 on page 29.

[8] We only allow the attacker to move forward in time, i.e. attackers having time machines are considered outside of our threat model.

Moreover, we grant the adversary access to all signatures created during the experiment, including the updated signatures created during epoch switches, by returning these signatures to the attacker. Furthermore, the adversary may explicitly request a signature for arbitrary excerpts. This models a scenario where the attacker might learn signatures from court proceedings, or as a regular customer of the cloud provider.

The adversary wins the experiment if (s)he manages to output a valid signature σ^* together with a forged excerpt E^* for some categories N^* of its choice. We want to exclude trivial cases from our definition, and therefore place restrictions on $\mathcal{A}$'s forgery, as in standard security notions.

Specifically, we require that if $\mathcal{A}$ obtained the secret key for some epoch t_{BreakIn}, then E^* must differ from the *real* excerpt for categories N^* with respect to an epoch $t < t_{\mathsf{BreakIn}}$. We formalize this by first "truncating" the log file M to the state of the most recent epoch switch (M'), then defining the *real* excerpt $E' := E(N^*, M')$ for the categories N^* chosen by the attacker. We then require that $\mathcal{A}$'s forgery E^* does not "agree" with E', i.e. that E' is not a prefix of the forged excerpt E^*. Any changes with regard to epochs $t \geq t_{\mathsf{BreakIn}}$ are trivial, since $\mathcal{A}$ obtained the key $\mathsf{sk}_{t_{\mathsf{BreakIn}}}$ and can therefore use the scheme's algorithms Append, $\mathsf{Extract}$ and Update to create regular (and hence valid) signatures.

As dicussed in Section 4.3, we currently do not know of any technique preventing an attacker from truncating the log file to the state of the most recent epoch switch in the standalone model. Hence, we consider any truncations up to the most recent epoch switch trivial. However, if an attacker could provide a valid signature for an excerpt E^* such that E^* is a prefix of E' (instead of the other way around) and $E^* \neq E'$, then this truncation would be considered non-trivial.

Our condition that E' must not be prefix of E^* also implicitly models the property of completeness. If an attacker managed to omit one or more log entries from E' in E^* (where the log entries were added before epoch t_{BreakIn}), then E' would not be a prefix of E^* and hence the forgery would be considered non-trivial. Only if no log entry from the epochs 1 through $t_{\mathsf{BreakIn}} - 1$ has been omitted is the attacker's output considered trivial.

As another restriction, we require that E^* was never returned to $\mathcal{A}$ as an excerpt for the categories N^*. If it was, then E^* used to be a correct excerpt for categories N^*, and the challenger created a signature σ^* for it. As discussed in Remark 4.11 on page 72, such attacks are always possible, and hence we consider the attack trivial. Note, however, that $\mathcal{A}$ may re-use an excerpt E returned by a query to the extraction oracle if $\mathcal{A}$ changes the set of associated categories N. I.e., if the attacker manages to have an excerpt E (for some categories N) accepted as an excerpt for another set of categories $N^* \neq N$, (and E differs from the correct excerpt with regard to an epoch before the break-in epoch) then $\mathcal{A}$'s forgery is considered non-trivial.

4.5. Our Scheme

We now describe a scheme that achieves the above security notion. We call it SALVE, for "Secure Audit Log with Verifiable Excerpts". The main ingredient for SALVE[9] is a forward-secure signature scheme. Let us briefly describe the basic ideas underlying our construction.

Sequence Numbers per Category.
Instead of adding only global sequence numbers, we augment signatures with sequence numbers (counters) c_ν for *each* category ν. In particular, the sequence numbers for the category "`All`" act as global sequence numbers.

Signing Counters.
Each log entry is signed along with the sequence numbers belonging to the categories of the log entry. All these counters are increased by one after the log entry has been signed. During verification, one checks if the counters of each category ν supposed to be present in the excerpt form the sequence $(1, \ldots, c_\nu)$. This way, one can detect duplicate log entries, log entries missing between present ones, and reordering attacks.

Epoch Markers with Counters.
Additionally, we sign all counters that have changed during an epoch i together with the epoch markers created at the end of epoch i. After these counters have been signed together with the epoch markers, the secret key is evolved using the Update algorithm.

4.5.1. Formal Description

We introduce some additional notation. When signing multiple counter values, we will sign a partial map $f\colon \{0,1\}^* \to \mathbb{N}$, which is formally modelled as a set f of pairs $(\nu, c_\nu) \in \{0,1\}^* \times \mathbb{N}$, signifying that the counter value of category ν is c_ν, or $f(\nu) = c_\nu$. For each category name ν, there is at most one pair in f that has ν as the first component. We also write such partial maps as $\{\nu_1 \mapsto c_{\nu_1}, \ldots, \nu_n \mapsto c_{\nu_n}\}$. A *key* of f is a bit string $\nu \in \{0,1\}^*$ for which $f(\nu)$ is defined. The set of keys for f is $\mathrm{keys}(f) \coloneqq \{\nu \in \{0,1\}^* \colon \exists c \in \mathbb{N} : (\nu, c) \in f\}$.

Recall our convention (introduced in Section 2.4), that we may sign mathematical objects $o \notin \{0,1\}^*$, for example a tuple (f, m') for a partial map f and a bit string $m' \in \{0,1\}^*$. These objects are implicitly mapped to bit strings using some encoding function before signing. We re-emphasize that we assume this encoding to be uniquely decodable, i.e. injective, see Section 2.4. This assumption is required for our proof of security.

[9] "This is what passes for humour amongst cryptographers." [AP13]

SALVE.

Let $\mathsf{FS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{Sign}, \mathsf{Verify})$ be a key-evolving signature scheme. The key-evolving categorized audit log scheme SALVE is given by the following algorithms:

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk}, \sigma_{0,1})$
: creates a key pair by running $(\mathsf{sk}_1, \mathsf{pk}) \leftarrow \mathsf{FS.KeyGen}(1^\kappa, 1^T)$. The initial signature is the empty sequence $\sigma_{1,0} := ()$. The output is $(\mathsf{sk}_1, \mathsf{pk}, \sigma_{1,0})$.

$\mathsf{Append}(\mathsf{sk}_t, M_{1,i-1}, \sigma_{1,i-1}, m_i = (N_i, m_i')) \to \sigma_{1,i}$
: is called to create a new signature $\sigma_{1,i}$ when a new log entry $m_i = (N_i, m_i')$ is appended to the current log file $M_{1,i-1} = (m_1, \ldots, m_{i-1})$. Besides $M_{1,i-1}$ and m_i, it also receives the current secret key sk_t and the current signature $\sigma_{1,i-1}$ as input.

  We assume "`EM`" $\notin N_i$, except when Append is called from the Update algorithm (see below), and "`All`" $\in N_i$.

  Append first determines the current counter values c_ν for all $\nu \in N_i$ (the total count of all log entries previously added to these categories).[10] Let $c_\nu := 0$ if the category ν has never occurred before.

  Next, Append creates the partial map $f_i = \{\nu \mapsto c_\nu + 1 \colon \nu \in N_i\}$, computes $\sigma_i' := \mathsf{FS.Sign}(\mathsf{sk}_t, (f_i, m_i'))$, and appends $\sigma_i := (f_i, \sigma_i')$ to $\sigma_{1,i-1}$ to obtain $\sigma_{1,i} := (\sigma_1, \ldots, \sigma_{i-1}, \sigma_i)$. It outputs $\sigma_{1,i}$.

$\mathsf{Update}(\mathsf{sk}_t, M_{1,i-1}, \sigma_{1,i-1}) \to \mathsf{sk}_{t+1}$
: is called at the end of each epoch t with the current secret key sk_t, the current log file $M_{1,i-1}$ and the current signature $\sigma_{1,i-1}$. It has two tasks: it must append an epoch marker to $M_{1,i-1}$ (and its accompanying signature to $\sigma_{1,i-1}$) and update the secret key.

  In order to create the epoch marker, it determines the set N of all categories that have received a new log entry during epoch t and the total number of log entries c_ν in each of these categories (including log entries from previous epochs).[11] It then creates the set of all these counters $f_i' := \{\nu \mapsto c_\nu \colon \nu \in N\}$ and encodes ("`End of epoch:`", t, f_i') =: m_i' as a bit string m_i' in some unique fashion. The epoch marker (which is a categorized log entry) is set to $m_i := (\{\text{"}\texttt{All}\text{"}, \text{"}\texttt{EM}\text{"}\}, m_i')$ and appended to $M_{1,i-1}$. Next, the Update algorithm computes a signature $\sigma_{1,i} := \mathsf{Append}(\mathsf{sk}_t, M_{1,i-1}, \sigma_{1,i-1}, m_i)$ for the log file including the epoch marker m_i.

  Finally, if $t < T$, Update computes $\mathsf{sk}_{t+1} := \mathsf{FS.Update}(\mathsf{sk}_t)$, securely erases sk_t and outputs sk_{t+1}. Otherwise it deletes sk_t and outputs $\mathsf{sk}_{t+1} := \bot$.

$\mathsf{Extract}(\mathsf{sk}_t, M_{1,i}, \sigma_{1,i}, N) \to \sigma$
: is tasked to create a signature for the excerpt $E(N)$ of the log file $M_{1,i}$ and the signature $\sigma_{1,i} = (\sigma_1, \ldots, \sigma_i)$. We assume that we always have "`EM`" $\in N$.

[10] These counter values may be cached or determined by searching for the most recent log entry added to each category.

[11] Again, this information may be cached.

The signature mostly consists of the individual signatures for all log messages in the excerpt, including the epoch markers, but also contains a newly generated signature for the entire excerpt. More formally, let $K \coloneqq I(N, M_{1,i})$, $l \coloneqq \text{len}(K)$, $(k_1, \ldots, k_l) = K$.

Then Extract computes the signature $\sigma_E \coloneqq \mathsf{FS.Sign}(\mathsf{sk}_t, (N, E))$, and outputs $\sigma \coloneqq (\sigma_{k_1}, \ldots, \sigma_{k_l}, \sigma_E)$ as the signature for E.

$\mathsf{Verify}(\mathsf{pk}, N, E, \sigma) \to 0/1$

must check the correctness of the excerpt $E = ((N_1, m'_1), \ldots (N_l, m'_l))$ (with $l \in \mathbb{N}_0$) for the categories N based on the public key pk and the signature σ. Verify first checks if $\text{len}(E) + 1 = \text{len}(\sigma)$, and rejects the input (outputs 0 and exits) if this is not the case. Otherwise, let $\sigma = ((f_1, \sigma'_1), \ldots, (f_l, \sigma'_l), \sigma_E)$. We assume that we always have "`EM`" $\in N$. If "`EM`" $\notin N$, the signature is rejected as invalid.

The algorithm will use counters c'_ν for all categories $\nu \in N$ to keep track of the number of log entries in each category that already occured in the excerpt. These counters will be compared with the actual counters from the signatures. We omit the usual string notation in the index, i.e., we simply write c_{EM} and c_{All} instead of $c_{\text{"EM"}}$ and $c_{\text{"All"}}$.

As a first step, Verify initializes its counters $c'_\nu \coloneqq 0$ for all $\nu \in N$. If "`All`" $\notin N$, it also sets $c'_{\text{All}} \coloneqq 0$. It then performs the following checks for each entry $m_i = (N_i, m'_i) \in E$, in the order of increasing i:

- It checks whether the signature for the individual log entry is valid:

$$\mathsf{FS.Verify}((\mathsf{pk}, c'_{\text{EM}} + 1, (f_i, m'_i)), \sigma'_i) = 1, \tag{4.1}$$

- whether m_i belongs to one of the requested categories:

$$N_i \cap N \neq \emptyset, \tag{4.2}$$

- whether m_i's set of category names N_i is unchanged:

$$\text{keys}(f_i) = N_i, \text{ and} \tag{4.3}$$

- whether the counter values signed together with the message are as expected:

$$f_i(\nu) = c'_\nu + 1 \text{ for all } \nu \in N \cap N_i. \tag{4.4}$$

- If m_i is an epoch marker, i.e. "`EM`" $\in N_i$, then Verify decodes m'_i to reconstruct f'_i. It then checks whether

$$f'_i(\nu) = c'_\nu \text{ for all } \nu \in \text{keys}(f'_i) \cap N. \tag{4.5}$$

- If "`All`" $\notin N$, it checks whether

$$f_i(\text{“}\texttt{All}\text{”}) > c'_{\text{All}} \tag{4.6}$$

  and sets $c'_{\text{All}} \coloneqq f_i(\text{“}\texttt{All}\text{”})$.

If any of these checks fail, Verify outputs 0. If they pass, Verify increments c'_ν by one for all $\nu \in N \cap N_j$. The verification procedure then continues with the next i, until (including) $i = l$.

Finally, Verify checks whether

$$\mathsf{FS}.\mathsf{Verify}((\mathsf{pk}, c'_{\text{EM}} + 1, (N, E)), \sigma_E) \stackrel{?}{=} 1, \tag{4.7}$$

and outputs 1 if so, and 0 otherwise.

A few notes are in order here:

1. Firstly, observe that for all log entries m_i, the number of epoch markers c_{EM} in the log file (or an excerpt) before m_i is one less than the number t of the epoch in which m_i was signed.

2. Excerpts created by SALVE are signed with the most recent secret key available. The verification algorithm implicitly checks for truncation attacks by requiring that the entire excerpt was signed during epoch t, where t is the number of epoch markers found in the excerpt plus 1 (see (4.7)). Thus, the final signature σ_E serves as an implicit proof that the signer knows the key of epoch $c'_{\text{EM}} + 1$. Truncating a log file (or an excerpt) to an epoch before the break-in therefore requires forging a signature σ_E supposedly created with a previous secret key, and thus breaking the security of FS.

3. If the verification algorithm had the current epoch number t as an additional trusted input, it could also check whether $t = c'_{\text{EM}} + 1$. This would strengthen the verification algorithm considerably.

4. Generally, given an excerpt E for some set of categories N, it is easy to create an excerpt for a subset of these categories, or to add other categories to E. However, creating a valid signature σ for the new excerpt is hard, because the set of category names N is included in the signature $\sigma_E \coloneqq \mathsf{FS}.\mathsf{Sign}(\mathsf{sk}_t, (N, E))$. We view this as a feature, as it prevents an attacker from tampering with excerpts.

5. Much information required by the above algorithms (e.g. current counter values and the set of categories modified since the last epoch switch) can be cached by an implementation. This way, SALVE can be implemented very efficiently.

Before proving SALVE correct, we first give a few examples illustrating how SALVE operates.

Example 4.14 (Signing and Updating)**.** We return to our bank example. When the log file is created, the KeyGen algorithm creates a pair of keys $(\mathsf{sk}_1, \mathsf{pk})$ and initializes the signature $\sigma_{1,0} := ()$ for the empty log file $M_{1,0} = ()$.

Let $m_1 := (N_1 = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{customer id 1}\text{”}, \text{“}\texttt{account creation}\text{”}\}, m_1')$ be the first entry added to the log file. The new log file is $M_{1,1} = (m_1)$. The Append algorithm is called to create a signature for $M_{1,1}$.

It first determines the number of log entries in the categories $\nu \in N_1$ so far. Since there have been no log entries before, we have $c_{\text{All}} = c_{\text{customer id 1}} = c_{\text{account creation}} = 0$.

It therefore sets $f_1 := \{\text{“}\texttt{All}\text{”} \mapsto 1, \text{“}\texttt{customer id 1}\text{”} \mapsto 1, \text{“}\texttt{account creation}\text{”} \mapsto 1\}$, and stores $\sigma_1 := (f_1, \sigma_1' := \mathsf{FS.Sign}(\mathsf{sk}_1, (f_1, m_1')))$ as the individual signature for the log entry m_1. The signature for $M_{1,1}$ is (σ_1).

Now let $m_2 := (N_2 = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{customer id 1}\text{”}, \text{“}\texttt{deposit}\text{”}\}, m_2')$ be the second log entry. When this log entry is added to $M_{1,1}$, we get $M_{1,2} = (m_1, m_2)$.

Again, one needs to create a signature for m_2 (and the new log file $M_{1,2}$). In order to compute the signature for m_2, the Append algorithm determines the counter values $c_{\text{All}} = 1$, $c_{\text{customer id 1}} = 1$ and $c_{\text{deposit}} = 0$. These are transformed into $f_1 := \{\text{“}\texttt{All}\text{”} \mapsto 2, \text{“}\texttt{customer id 1}\text{”} \mapsto 2, \text{“}\texttt{deposit}\text{”} \mapsto 1\}$. The signature for m_2 is $\sigma_2 := (f_2, \mathsf{FS.Sign}(\mathsf{sk}_1, (f_2, m_2')))$. This is appended to $\sigma_{1,1}$ to obtain $\sigma_{1,2} = (\sigma_1, \sigma_2)$, the signature for $M_{1,2}$.

Now suppose there is an epoch switch from epoch 1 to epoch 2. The Update algorithm is called. It first collects the counter values of all categories that have had a log entry added to them in epoch 1. These counter values are $c_{\text{All}} = 2$, $c_{\text{customer id 1}} = 2$, $c_{\text{account creation}} = 1$, $c_{\text{deposit}} = 1$, and encodes them to

$$f_3' := \{\text{“}\texttt{All}\text{”} \mapsto 2, \text{“}\texttt{customer id 1}\text{”} \mapsto 2, \\ \text{“}\texttt{account creation}\text{”} \mapsto 1, \text{“}\texttt{deposit}\text{”} \mapsto 1\}.$$

It then encodes the tuple $(\text{“}\texttt{End of epoch:}\text{”}, 1, f_3')$ as a bit string m_3'. This bit string is converted to a categorized log message $m_3 := (N_3 = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{EM}\text{”}\}, m_3')$ by assigning it to the categories “All” and “EM”.

Next, m_3 is to be appended to the log file. The Update algorithm computes the new signature $\sigma_{1,3}$ as before: It determines the counter values $c_{\text{All}} = 2$, $c_{\text{EM}} = 0$, and sets $f_3 := \{\text{“}\texttt{All}\text{”} \mapsto 3, \text{“}\texttt{EM}\text{”} \mapsto 1\}$. It then creates the signature $\sigma_3' := \mathsf{FS.Sign}(\mathsf{sk}_1, (f_3, m_3'))$ and appends $\sigma_3 := (f_3, \sigma_3')$ to $\sigma_{1,2}$. The result is $\sigma_{1,3} = (\sigma_1, \sigma_2, \sigma_3)$. Observe that since m_3' contains f_3' and m_3' has been signed, the number of log entries in all categories is authenticated with sk_1.

Before Update terminates, it evolves sk_1 to $\mathsf{sk}_2 := \mathsf{FS.Update}(\mathsf{sk}_1)$, and securely erases sk_1.

Now assume that one adds two messages in epoch 2: The first one is $m_4 := (N_4 = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{customer id 2}\text{”}, \text{“}\texttt{account creation}\text{”}\}, m_4')$ and the second is $m_5 := (N_5 = \{\text{“}\texttt{All}\text{”}, \text{“}\texttt{customer id 1}\text{”}, \text{“}\texttt{withdrawal}\text{”}\}, m_5')$. The corresponding maps are

$$f_4 = \{\text{“}\texttt{All}\text{”} \mapsto 4, \text{“}\texttt{customer id 2}\text{”} \mapsto 1, \text{“}\texttt{account creation}\text{”} \mapsto 2\}, \text{ and}$$
$$f_5 = \{\text{“}\texttt{All}\text{”} \mapsto 5, \text{“}\texttt{customer id 1}\text{”} \mapsto 3, \text{“}\texttt{withdrawal}\text{”} \mapsto 1\}.$$

We skip to the next epoch switch, as the signatures σ_4 and σ_5 are created as above. At the epoch switch from epoch 2 to epoch 3, Update is called. It first constructs

$$f'_6 = \{\text{“}\texttt{All}\text{”} \mapsto 5, \text{“}\texttt{account creation}\text{”} \mapsto 2, \text{“}\texttt{customer id 1}\text{”} \mapsto 3, \\ \text{“}\texttt{customer id 2}\text{”} \mapsto 1, \text{“}\texttt{withdrawal}\text{”} \mapsto 1\}.$$

Observe that the counter for the category “deposit” is not contained in f'_6, since there was no log entry in that category during epoch 2. Update creates a categorized log message $m_6 = (\{\text{“}\texttt{All}\text{”}, \text{“}\texttt{EM}\text{”}\}, m'_6 = (\text{“}\texttt{End of epoch:}\text{”}, 2, f'_6))$, signs it (resulting in σ_6), and appends m_6 and σ_6 to the log file $M_{1,5}$ and the signature so far $\sigma_{1,5}$, respectively. It then computes $\mathsf{sk}_3 \coloneqq \mathsf{Update}(\mathsf{sk}_2)$, deletes sk_2 in an unrecoverable fashion and outputs sk_3.

Example 4.15 (Excerpts and Verification)**.** Say someone requested an excerpt for any log entries regarding customer 2. Then one creates an excerpt for the categories $N = \{\text{“}\texttt{customer id 2}\text{”}, \text{“}\texttt{EM}\text{”}\}$. (Recall that by convention, we have “EM” $\in N$ when the extraction algorithm is called.)

The excerpt to be output is $E \coloneqq (m_3, m_4, m_6)$, since $m_3, m_6 \in C(\text{“}\texttt{EM}\text{”})$ and $m_4 \in C(\text{“}\texttt{customer id 2}\text{”})$. Thus, the signature σ for E contains σ_3, σ_4 and σ_6. The last component of σ is a signature σ_E for (N, E).

The verification algorithm gets (m_3, m_4, m_6) and $(\sigma_3, \sigma_4, \sigma_6, \sigma_E)$ as input, along the public key pk. It verifies whether σ_3, σ_4 and σ_6 are valid for m_3, m_4, m_6 using $\mathsf{FS.Verify}$. Note that all epoch markers are included in the excerpt, so Verify can determine the epoch in which these messages were signed by counting the number of epoch markers occuring before the respective message. (In our description above, this is just c'_{EM}.)

The verification algorithm also checks whether $\text{keys}(f_j) = N_j$. To understand this, observe that N_j is not signed directly during the signature algorithm, but implicitly (since f_j is signed). If one omitted this check, an adversary might tamper with the categories N_j of the excerpt without the verification algorithm detecting this.

Verify also checks that all counters in f_j match the expected values. As a last step, Verify checks the signature over the entire excerpt E together with the set of categories N for which this excerpt was created. For this check, it determines the epoch number based on the number of epoch markers in the excerpt.

The last component σ_E of a signature σ serves two purposes: Firstly, it is necessary to prevent attackers not having a secret key from freely “combining” signatures for different excerpts. For example, without the additional signature over all log entries in E, if an attacker had signatures for excerpts for the categories N_1 and N_2, then it were trivial for the adversary to create a signed excerpt for $N_1 \cup N_2$ or $N_1 \cap N_2$. Secondly, this signature implicitly proves the signer knows sk_3, and thus that the excerpt has not been truncated.

In comparison, if the signer could only sign (N, E) with, say, sk_5, then it would be evident that the excerpt is missing at least the epoch markers indicating the end of epochs 3 and 4. While this alone might not seem harmful for the verification of the excerpt, observe that without these epoch markers, the verifier has no way to tell if new log entries were added to the category “customer id 2” during these epochs. The

correct excerpt generated with sk_5 has to contain these epoch markers, so the verifier can check if the counter value for this category has increased during the respective epoch. If no updated counter value is contained in the respective epoch markers, then the verifier can be certain that no log entry was added to the category during these epochs. Without the epoch markers, however, the verifier might fall victim to a truncation.

This concludes our examples for SALVE. We now show:

Lemma 4.16. SALVE is correct if FS is correct.

Proof. We need to show that all checks of Verify pass, when Verify is called with a regularly created signature $\sigma = (\sigma_1, \ldots, \sigma_l, \sigma_E)$.

First let us gather some simple observations:

1. Verify correctly counts the number of entries it has seen for each category $\nu \in N$ as c'_ν. The sequence number expected to be found in the next log message belonging to category ν is $c'_\nu + 1$.

2. In particular, c'_{EM} contains the number of epoch markers it has encountered so far, which is one less than the epoch during which the next message should have been signed (see Note 1 on page 79).

3. Similar to observation 1, c'_{All} is the maximum sequence number in the category "`All`" that Verify has encountered.

Now let us show that the checks of Verify pass. For each $i \in [l]$, check (4.1) will pass due to the correctness of FS, and because of observation 2.

Check (4.2) will always hold true, because Extract only considers messages that are contained in the excerpt, see Definition 4.7. Check (4.3) will pass, too, because of the construction of f_i in the Append algorithm.

Check (4.4) will pass since for each $\nu \in N_i$, Append has set $f_i(\nu)$ to one plus the number of log entries contained in category ν, all of these entries are contained in the excerpt, and Verify counts these (as c'_ν) correctly. A similar argument shows that check (4.5) is successful.

If "`All`" $\notin N$, check (4.6) verifies that the counters for the category "`All`" that are signed together with each log entry form a strictly increasing sequence. (If "`All`" $\in N$, this is already verified by check (4.4). Furthermore, check (4.4) also verifies that the counter values are consecutive.) This is always the case for excerpts created by the regular mechanism, so this check will never fail.

Finally, equation (4.7) will hold because of the correctness of FS. Hence, for a regular signature all checks pass, and Verify outputs 1. □

4.5.2. Security Analysis

We now analyze the security of SALVE. The following theorem states our main result:

Theorem 4.17 (Security of SALVE)**.** Assume that SALVE uses an injective encoding to map the signed objects to bit strings. If there exists a PPT attacker $\mathcal{A}$ that wins the FS-EUF-CLMA experiment against SALVE with probability $\varepsilon_{\mathcal{A}}$, then there exists a PPT attacker $\mathcal{B}$ that wins the FS-EUF-CMA game against FS with probability $\varepsilon_{\mathcal{B}} = \varepsilon_{\mathcal{A}}$.

Proof. Let $\mathcal{A}$ be an attacker having success probability $\varepsilon_{\mathcal{A}}$ in the FS-EUF-CLMA experiment against SALVE. We construct an adversary $\mathcal{B}$ which tries to break the FS-EUF-CMA-security of the underlying scheme FS, using $\mathcal{A}$ as a component. $\mathcal{B}$ must simulate the FS-EUF-CLMA experiment with SALVE for $\mathcal{A}$. $\mathcal{B}$ does this as follows.

$\mathcal{B}$ receives a public key pk and the number of epochs T as input. It sets $t \coloneqq 1$, $i \coloneqq 1$, $M_{1,0} \coloneqq ()$, $\sigma_{1,0} \coloneqq ()$. It then starts executing $\mathcal{A}$ with input $(\mathsf{pk}, 1^T, \sigma_{1,0})$.

When $\mathcal{A}$ issues an oracle query, $\mathcal{B}$ reacts as follows:

Signature Queries.

When $\mathcal{A}$ requests that a new message $m_i = (N_i, m_i')$ shall be added to the log file, $\mathcal{B}$ first verifies that "`All`" $\in N_i$ but "`EM`" $\notin N$. If this is not the case, then $\mathcal{B}$ aborts.

Otherwise, $\mathcal{B}$ collects the counter values c_ν for all $\nu \in N_i$, initializing them to 0 if the category ν has not occured before. It builds $f_i \coloneqq \{\nu \mapsto c_\nu + 1 \colon \nu \in N_i\}$ and submits (f_i, m_i') to the signature oracle in the FS-EUF-CMA experiment. This oracle answers with a signature σ_i' for (f_i, m_i'). $\mathcal{B}$ combines this with f_i to get $\sigma_i \coloneqq (f_i, \sigma_i')$. Then $\mathcal{B}$ sets $\sigma_{1,i} \coloneqq \sigma_{1,i-1} \parallel \sigma_i$, $M_{1,i} \coloneqq M_{1,i-1} \parallel m_i$, returns $\sigma_{1,i}$ to $\mathcal{A}$, and increments $i \coloneqq i + 1$.

Epoch Switching Queries.

When $\mathcal{A}$ requests an epoch switch from epoch t to epoch $t + 1$ in the FS-EUF-CLMA experiment, $\mathcal{B}$ verifies that $t < T$ and aborts if this is not the case. Then $\mathcal{B}$ creates the epoch marker just as in the Update algorithm: It first determines the set N of categories that had a log entry added to them during epoch i, collects the counters c_ν for all $\nu \in N$, builds $f_i' \coloneqq \{\nu \mapsto c_\nu \colon \nu \in N\}$ and sets $m_i' \coloneqq$ ("`End of epoch:`", t, f_i'). It then simulates the Append algorithm for $m_i \coloneqq (\{$"`All`", "`EM`"$\}, m_i')$ as described above and obtains a signature σ_i for m_i. The signature σ_i is added to $\sigma_{1,i-1}$, m_i is added to $M_{1,i-1}$, and i is incremented as above.

Finally, it calls the epoch switching oracle in the FS-EUF-CMA experiment, and increments $t \coloneqq t + 1$. It returns $M_{1,i}$ and $\sigma_{1,i}$ to $\mathcal{A}$.

Excerpt Queries.

When $\mathcal{A}$ requests a signature for an excerpt for the categories N, $\mathcal{B}$ proceeds as follows. If "`EM`" $\notin N$, then $\mathcal{B}$ aborts. Otherwise $\mathcal{B}$ first builds $E(N, M_{1,i})$. Next, $\mathcal{B}$ collects the individual signatures σ_j for all $m_j \in E$. (More formally, let $l = \mathrm{len}(E)$, and let $I(N, M_{1,i}) = (j_1, \ldots, j_l)$ denote the index sequence of the excerpt E with respect to $M_{1,i}$.) $\mathcal{B}$ submits (N, E) to the signature oracle in

the FS-EUF-CMA experiment to obtain σ_E. It sets $\sigma = (\sigma_{j_1}, \ldots, \sigma_{j_l}, \sigma_E)$ and returns (E, σ) to $\mathcal{A}$.

Breaking In.
When $\mathcal{A}$ requests the current secret key sk_t in the FS-EUF-CLMA experiment, $\mathcal{B}$ obtains it from its own oracle in the FS-EUF-CMA experiment and passes it to $\mathcal{A}$.

It is easy to see that the joint distribution of all values occuring in $\mathcal{B}$'s simulation of the FS-EUF-CLMA experiment ($\mathcal{A}$'s "view") matches the distribution in the real FS-EUF-CLMA experiment.

At the end of the experiment, $\mathcal{A}$ outputs a forged excerpt E^*, a set of categories N^* and a forged signature σ^* for E^*. If $\mathcal{A}$ outputs an invalid or trivial forgery, then $\mathcal{B}$ outputs $\perp$ and aborts. Otherwise, $\mathcal{B}$ determines which of the following cases has occured and acts as described for each case. For this distinction, we let c^*_{EM} be the number of log entries (N^*_j, m'^*_j) in E^* with "$\mathtt{EM}$" $\in N^*_j$.

Case 1: E^* contains c^*_{EM} epoch markers with $c^*_{\mathrm{EM}} + 1 < t_{\mathsf{BreakIn}}$.
Note that this case also captures the event that $\mathcal{A}$ does not obtain a secret key at all (because then $t_{\mathsf{BreakIn}} = \infty$). In this case, $\mathcal{B}$ outputs $m^* := (N^*, E^*)$ as its message, the number $t^* := c^*_{\mathrm{EM}} + 1$ as the epoch number, and the last element σ^*_E of the sequence σ^* as its forged signature for m^*. σ^*_E must be a valid signature for (N^*, E^*), since otherwise Verify would have rejected the signature σ^* after checking (4.7).

All queries that $\mathcal{B}$ submitted to its signature oracle during epoch t^* (if any) were either of the form (f_j, m'_j) for some messages (including epoch markers) $m_j = (N_j, m'_j)$ or of the form (N, E) for extraction queries. Because of the uniqueness of the encoding, all of $\mathcal{B}$'s signature queries (f_j, m'_j) for log messages (N_j, m'_j) differ from (N^*, E^*). Also, since E^* is a non-trivial forgery in the FS-EUF-CLMA game, $\mathcal{B}$ did never request a signature for (N^*, E^*). Finally, since $t^* < t_{\mathsf{BreakIn}}$, $\mathcal{B}$'s output is a non-trivial forgery in the FS-EUF-CMA experiment.

Hence, $\mathcal{B}$'s output is valid and non-trivial, so $\mathcal{B}$ wins the FS-EUF-CMA game.

Case 2: E^* contains c^*_{EM} epoch markers with $c^*_{\mathrm{EM}} + 1 \geq t_{\mathsf{BreakIn}}$.
Let M' and E' be as in Definition 4.13, that is, M' is the log file returned by $\mathcal{A}$'s most recent call to the epoch switching oracle, and E' is the excerpt for the categories N^* of M'. Observe that if $\mathcal{A}$ broke in during epoch $t_{\mathsf{BreakIn}} = 1$, then we had $M' = ()$ by definition, and so $E' = ()$, which is a prefix of *all* excerpts E^* that $\mathcal{A}$ may have created. Thus, any forgery of $\mathcal{A}$ were trivial, and $\mathcal{A}$ could not win the game. In the following, we may therefore assume $t_{\mathsf{BreakIn}} > 1$.

Let E'^* be the prefix of E^* up until (including) the $(t_{\mathsf{BreakIn}} - 1)$-th epoch marker (the $(t_{\mathsf{BreakIn}} - 1)$-th log message $(N^*_i, m'^*_i) \in E^*$ with "$\mathtt{EM}$" $\in N^*_i$). We know that E' is not a prefix of E'^*, since otherwise E' would also be a prefix of E^* in contradiction to $\mathcal{A}$'s forgery not being trivial.

Let $l = \text{len}(E')$, $l^* = \text{len}(E'^*)$, $E' = (m_i)_{i=1}^{l}$, $E'^* = (m_i^*)_{i=1}^{l^*}$, $m_i^* = (N_i^*, m_i'^*)$ for all $i \in [l^*]$ and $m_i = (N_i, m_i')$ for all $i \in [l]$. $\mathcal{B}$ builds the sequences $S^* = ((f_1^*, m_1'^*), \ldots, (f_{l^*}^*, m_{l^*}'^*))$ (taking the f_i^* from the signatures $\sigma_i^* \in \sigma^*$) and $S = ((f_1, m_1'), \ldots, (f_l, m_l'))$ (taking the f_i from the signatures σ_i it constructed during the simulation). Note that S contains exactly $\mathcal{B}$'s oracle queries during epochs 1 through $t_{\mathsf{BreakIn}} - 1$, restricted to those messages that belong to at least one of the categories N^*. Also observe that $S^* \neq S$, since we otherwise had $E'^* = E'$ (by (4.3)) in contradiction to E' not being a prefix of E'^*.

The key observation is that there must be a $(f_j^*, m_j'^*) \in S^*$ with $(f_j^*, m_j'^*) \notin S$ ($j \in [l^*]$). Suppose for the sake of a contradiction that there is no such pair. Then S^* consists entirely of pairs that also occur in S. Obviously, S^* can not contain duplicate pairs $(f_j^*, m_j'^*)$, since the verification algorithm would have rejected the excerpt when checking that counters always increase (equations (4.4) and/or (4.6)). Since S^* contains only pairs also contained in S, contains no duplicates, and $S^* \neq S$, S^* is missing at least one tuple from S. Since S^* (by construction) contains exactly $t_{\mathsf{BreakIn}} - 1$ epoch markers, S^* is missing a log entry which is not an epoch marker. But then Verify had failed when checking the counters in (4.5), which is impossible if $\mathcal{A}$'s output was valid.

So we have established that S^* contains a pair $(f_j^*, m_j'^*) \notin S$. $\mathcal{B}$ searches for this pair, and outputs it as its message. It also outputs one plus the number of epoch markers before $(f_j^*, m_j'^*)$ as the epoch number t^* and $\sigma_j'^*$ as the signature.

This is a valid signature in the FS-EUF-CMA experiment, since (4.1) holds. It remains to show that this is a non-trivial forgery. Firstly, the number of epoch markers before $(f_j^*, m_j'^*)$ is at most $t_{\mathsf{BreakIn}} - 2$, since E'^* contains at most $t_{\mathsf{BreakIn}} - 1$ epoch markers, and the last log message in E'^* is one of these. Hence, the number of epoch markers *before* $(f_j^*, m_j'^*)$ is at most $t_{\mathsf{BreakIn}} - 2$. Therefore, the signature $\sigma_j'^*$ is valid for an epoch $t^* \leq t_{\mathsf{BreakIn}} - 1 < t_{\mathsf{BreakIn}}$. Secondly, observe that m_j^* belongs to at least one of categories N^*, since Verify had rejected the excerpt when checking (4.2) for m_j^* otherwise. Thus, $\mathcal{B}$ has never requested $(f_j^*, m_j'^*)$ from its signature oracle, since $(f_j^*, m_j'^*) \notin S$, where S is exactly the sequence of $\mathcal{B}$'s signature queries for all messages belonging to at least one of the categories N^*, such as m_j^*. Also, all of $\mathcal{B}$'s signature queries of the form (N, E) differ from $(f_j^*, m_j'^*)$, since we assumed that their encoding to bit strings is injective. Hence, $\mathcal{B}$ wins the FS-EUF-CMA game in case 2, since it outputs a non-trivial and valid forgery.

Since $\mathcal{B}$'s simulation of the FS-EUF-CLMA game for $\mathcal{A}$ is perfect, $\mathcal{B}$ wins both in case 1 and in case 2, and one of these cases occurs whenever $\mathcal{A}$ outputs a valid and non-trivial signature, we have $\varepsilon_{\mathcal{B}} = \varepsilon_{\mathcal{A}}$. Also, $\mathcal{B}$ runs in polynomial time, as $\mathcal{A}$ does. □

Corollary 4.18. If FS is FS-EUF-CMA-secure, and SALVE uses injective encodings, then SALVE is FS-EUF-CLMA-secure.

This concludes our treatment of SALVE's ability to protect the authenticity and integrity of excerpts, as well as verify the completeness of excerpts.

Remark 4.19 (Enhanced Confidentiality)**.** Note that while SALVE can create excerpts and thus protect the confidentiality of log records not in contained in the excerpts, each excerpt leaks the *number* of log records added to each category during each epoch. Formalizing a security notion regarding the confidentiality of this information as well as designing a scheme provably attaining such a notion is outside of the scope of this thesis and is left as future work.

4.5.3. Performance Analysis

In this section, we analyze the runtime and storage overhead of SALVE. Our findings are derived from the algorithms described in Section 4.5.1. Since SALVE can be instantiated with an arbitrary forward-secure signature scheme FS, we give our findings with regard to algorithm runtime in terms of calls to algorithms of FS, and our findings in regard to storage overhead in terms of key and signature sizes of FS, respectively. Table 4.1 summarizes our findings.

Throughout our analysis, let M denote the current log file, l be the number of log entries in M (not counting epoch markers, if any), t be the current epoch, $m_i = (N_i, m'_i)$ be the log message added during Append, R be the total number of associations between log entries (excluding epoch markers) and categories (i.e. $R := \sum_{j=1}^{l} |N_j|$), E be the excerpt being signed by the $\mathsf{Extract}$ algorithm or verified by Verify, l' be the number of log entries in E (again, not counting epoch markers), N the set of requested categories for an excerpt, N_{total} be the set of (the names of) all categories that have been used so far, and N_{epoch} be the set of (the names of) the categories that have received a new log entry in the epoch being ended by the update procedure. Our runtime analysis assumes that:

- All sequence numbers c_ν and category names ν have size $\mathcal{O}(1)$, i.e. there is an a-priori-bound on the length of these. We stress that we make this assumption purely to simplify the analysis. Our scheme can handle sequence numbers and category names of arbitrary length.

- The implemenation always stores sets N of category names in an ordered fashion in order to achieve a unique representation. Maps f_i are ordered as well, by N_i.

- The implementation caches sequence numbers in balanced binary trees. In this case, lookup, insertion and update operations to the cache take $\mathcal{O}(\log|N_{\text{total}}|)$ time units. This is a conservative assumption, since the same operations have an expected cost of $\mathcal{O}(1)$ time units for caches based on hash tables.

- The implementation caches the names of all categories which have received a new log message in the current epoch, i.e. N_{epoch}.

- We also assume that encoding and decoding pairs (f_j, σ'_j) to and from $\{0,1\}^*$ takes time $\mathcal{O}(|f_j| + \text{len}(\sigma'_j))$.

Table 4.1.: Performance characteristics of SALVE in relation to FS. We use sets, sequences and bit strings instead of their size and length, respectively, to relieve notation.

Algorithm	Runtime
KeyGen	$1 \times \mathsf{FS.KeyGen} + \mathcal{O}(1)$
Append	$1 \times \mathsf{FS.Sign} + \mathcal{O}(N_i(\log N_i + \log N_{\text{total}}) + m'_i)$
Update	$1 \times \mathsf{FS.Update} + 1 \times \mathsf{FS.Sign} + \mathcal{O}(N_{\text{epoch}} \log N_{\text{total}})$
Extract	$1 \times \mathsf{FS.Sign} + \mathcal{O}(R \log N + t)$
Verify	$(l' + t) \times \mathsf{FS.Verify} + \mathcal{O}((R + t) \log N)$

Datum	Size
Secret Key	$1 \times \mathsf{sk_{FS}} + 0$
Public Key	$1 \times \mathsf{pk_{FS}} + 0$
Log File Signature	$(l + t - 1) \times \sigma_{\mathsf{FS}} + \mathcal{O}(R + t)$
Excerpt Signature	$(l' + t) \times \sigma_{\mathsf{FS}} + \mathcal{O}(R + t)$

Algorithm Runtime Analysis

Key Generation.
The runtime of the KeyGen algorithm is dominated by the call to FS.KeyGen, which creates a key for T time periods. All other computations can be done in $\mathcal{O}(1)$ time units.

Message Signing.
The Append algorithm must determine the current counter values c_ν for all $\nu \in N_i$ in order to create the mapping f_i. We assume that the algorithm first sorts N_i in order to achieve a unique representation. This can be done in $\mathcal{O}(|N_i| \log |N_i|)$ time units. Looking up all counter values takes $\mathcal{O}(|N_i| \log |N_{\text{total}}|)$ time units. Encoding (f_i, m_i') to a binary string takes time $\mathcal{O}(|f_i| + \text{len}(m_i')) = \mathcal{O}(|N_i| + \text{len}(m_i'))$. The signing of the tuple then takes one call to FS.Sign. Updating the cached sets N_{total} and N_{epoch} requires time $\mathcal{O}(|N_i|(\log|N_{\text{total}}| + \log|N_{\text{epoch}}|)) \subset \mathcal{O}(|N_i| \log|N_{\text{total}}|)$.

Updating the Secret Key.
The Update algorithm accesses the cached set N_{epoch} and looks up the corresponding counter values c_ν. This takes at most $\mathcal{O}(|N_{\text{epoch}}| \log|N_{\text{total}}|)$ time units. It then calls the Append algorithm, and thus inherits its runtime costs. Note that N_i is constant for this call, so $|N_i| = 2$ can be disregarded in the $\mathcal{O}$ notation. Finally, it performs a call to FS.Update.

Extraction of Excerpts.
Extract first sorts N in time $\mathcal{O}(|N| \log|N|)$. It then scans through M to find relevant log entries. For each log entry $m_i = (N_i, m_i')$, the algorithm can check if $N_i \cap N = \emptyset$ with at most $|N_i|$ lookup operations in N. Thus, scanning the entire log file takes $\mathcal{O}(\sum_{i=1}^{l} |N_i| \log|N| + t) = \mathcal{O}(R \log|N| + t)$ time units.

Verification.
The verification algorithm takes $l' + (t-1) + 1 = l' + t$ calls to FS.Verify for checks (4.1) and (4.7).

Checks (4.2) and (4.4) take $\mathcal{O}(|N_i| \log|N|)$ operations per iteration, check (4.3) only $\mathcal{O}(|N_i|)$. Check (4.6) can be done in $\mathcal{O}(|f_i|) = \mathcal{O}(|N_i|)$ time units.

For check (4.5), let $N_{\text{epoch},i}$ be the set of categories that received at least one new entry during epoch i. Then all checks of this type can be implemented in time $\mathcal{O}(\sum_{i=1}^{t} |N_{\text{epoch},i}| \log|N|)$.

In total, we have $(l' + t)$ calls to FS.Verify, and

$$\begin{aligned}
&\mathcal{O}\left(\sum_{i=1}^{l} |N_i| \log|N| + \sum_{i=1}^{t} |N_{\text{epoch},i}| \log|N|\right) \\
&= \mathcal{O}\left(\left(\sum_{i=1}^{l} |N_i| + \sum_{i=1}^{t} |N_{\text{epoch},i}|\right) \log|N|\right) \\
&= \mathcal{O}((R+t) \log|N|)
\end{aligned}$$

other operations.

Storage Overhead

In the following, we analyze the storage overhead imposed by SALVE.

Key Sizes.
The sizes of SALVE's public and secret keys are the same as FS's.

Log File Signature Size.
A signature for a log file M consists of $l + (t-1)$ signatures of FS, as well as the maps f_i, which take $\mathcal{O}(\sum_{i=1}^{l} |N_i| + t) = \mathcal{O}(R + t)$ bits.

Excerpt Signature Size.
The signature for an excerpt E consists of each log entry's individual signature, including the signatures for all epoch markers, and a final signature on the pair (N, E). We thus have $l' + (t-1) + 1 = l' + t$ signatures of FS. Furthermore, we have $(l' + (t-1))$ maps f_i, which take at most $\mathcal{O}(R + t)$ bits in total.

Comparison to Other Schemes

We now compare the efficiency of SALVE to the performance of other schemes in the literature. In particular, we compare to the scheme by Ma and Tsudik [MT08; MT09] and the Logcrypt scheme by Holt [Hol06], since both constructions are generically built on an underlying signature scheme, too. We also compare SALVE to the BAF [YP09; YPR12a] scheme by Yavuz et al.

However, Ma and Tsudik require a signature scheme that is not only forward-secure, but can also sequentially aggregate signatures, while Holt's scheme uses a *standard* digital signature without special properties such as forward security or sequential aggregation.[12] SALVE can be seen in between these two, as SALVE requires the underlying signature scheme to be forward-secure, but does not require the aggregation property.

The different requirements on the underlying signature scheme make it very hard to compare these schemes fairly. For example, the aggregate signature scheme used by Ma and Tsudik hides the amount of work required to verify a signature behind just one call to the aggregate verification algorithm. Comparison is complicated further by the issue that both Ma and Tsudik as well as Holt propose to perform an epoch switch every time a log entry has been added. (This is a case in which SALVE performs badly. However, given the linear overheads imposed by Logcrypt and Ma's and Tsudik's schemes, their schemes are not very practical in this case, neither.)

Comparing these three schemes to BAF is even harder, since BAF is not generically built on an arbitrary signature scheme (possibly requiring additional properties), but uses very concrete hardness assumptions and constructions.

[12] Holt implicitly constructs a forward-secure scheme from it by building a long certification chain which is embedded in the log file. The forward-secure scheme is a simple variant of the "Long Signature" scheme from [BM99, Section 2].

Table 4.2 shows our results. For Logcrypt, SALVE, and the scheme by Ma and Tsudik, KeyGen, Update, Sign, AggSign, and Verify refer to the costs to call the respective underlying signature scheme's algorithm. Similarly, $\text{len}(\mathsf{sk}), \text{len}(\mathsf{pk}), \text{len}(\sigma)$ refer to the sizes of the underlying scheme's secret key, public key and signatures, respectively. For Logcrypt, $n \in \mathbb{N}$ is a parameter that can be chosen freely. For BAF, ModExp, ModMul and ModAdd refer to the costs of modular exponentiation, multiplication and addition respectively, and H refers to the cost of evaluating a hash function on a relatively short input. BigInt refers to the size of a large integer value.[13]

Comparison with Logcrypt and the MT scheme. We see that SALVE is competitive with Logcrypt and the scheme by Ma and Tsudik in terms of key generation time, log entry signing time, as well as secret and public key size. It performs only slightly worse than these schemes for the key evolution and verification algorithms. (All forward-secure sequential aggregate signature schemes that we know of require at least $\mathcal{O}(\text{len}(M))$ operations. These operations may be modular squarings or even pairing evaluations.)

In terms of storage overhead for the log file SALVE beats Logcrypt, but cannot level with the scheme by Ma and Tsudik, since they use (sequential) *aggregate* signatures.

Note that the aggregation approch by Ma and Tsudik comes with two severe drawbacks: Firstly, their scheme cannot verify any log entry individually without verifying the entire log file. Secondly, if a single log entry is modified, verification of the entire log file fails, and *all* information stored in the log file must be considered to be forged by the adversary. Ma and Tsudik recognize these drawbacks, and devise an alternative "immutable" scheme that solves these issues. The modified scheme has $(\text{len}(M) + 1) \times \text{len}(\sigma)$ storage overhead, which is notably but not far better than SALVE.[14]

Comparison with BAF. As stated before, comparing SALVE to BAF is very hard, since SALVE may have very different performance characteristics depending on the underlying signature scheme FS.

BAF is heavily optimized for an efficient signing procedure. It also has an efficient key evolution algorithm, a modest secret key size and a very compact signature, that is independent of $\text{len}(M)$, just as the scheme by Ma and Tsudik. (BAF therefore carries the same drawbacks.) These enjoyable performance properties of BAF are paid for with a very expensive key generation algorithm and an extreme public key size. In comparison, while it is unlikely that SALVE could beat BAF with respect to signing and updating time, SALVE has more balanced performance properties overall.

[13] BAF uses prime-order subgroups of a prime field where the discrete logarithm problem is intractable with current methods and equipment. In order not to complicate our analysis further, we do not differentiate between integers in the size of the group order (at least 160 bits) and integers in the size of the prime field size (at least 1024 bits). One may conservatively assume that all of these integers are 160 bits in size, referring only to the group order.

[14] We show how to construct a scheme that can tolerate some modifications to the log file while still having sub-linear storage overhead in Chapter 5.

Table 4.2.: Comparison of SALVE with other Secure Logging Schemes.

Algorithm	Logcrypt	SALVE	Ma and Tsudik	BAF
		Runtime		
Key Generation	KeyGen	KeyGen	KeyGen	$2T \times \text{ModExp} + 5T \times H$
Log Entry Signing	Sign	Sign	AggSign	$2 \times H + 2 \times \text{ModAdd}$
Updating	KeyGen $+1/n \times$ Sign[15]	Update + Sign	Update	$2 \times H$
Excerpt Signing	—	Sign	—	—
Verification	$l \times$ Verify	$(l' + t) \times$ Verify	Verify	$(l+1) \times \text{ModExp} +$ $(2l-1) \times \text{ModMul}$

Datum	Size			
Secret Key	$\mathcal{O}(n) \times \text{len}(\mathsf{sk})$	$\text{len}(\mathsf{sk})$	$\text{len}(\mathsf{sk})$	$4 \times \text{BigInt}$
Public Key	$\text{len}(\mathsf{pk})$	$\text{len}(\mathsf{pk})$	$\text{len}(\mathsf{pk})$	$(4T+3) \times \text{BigInt}$
Log File Signature	$(l+t) \times \text{len}(\sigma) + t \times \text{len}(\mathsf{pk})$	$(l+t-1) \times \text{len}(\sigma)$	$\text{len}(\sigma)$	$2 \times \text{BigInt}$
Excerpt Signature	—	$(l' + t) \times \text{len}(\sigma)$	—	—

4.6. Summary

The ability to verify excerpts can be useful (i) to provide full confidentiality and privacy of most of the log entries, even when a subset of the log entries needs to be disclosed, (ii) to save resources during transmission and storage of the excerpt, and (iii) to ease manual review of log files.

We have defined a security notion for logging schemes which can provide excerpts. This security notion includes the property of verifiable *completeness*, i.e. the logging scheme's ability to detect omissions. Moreover, the security notion also considers truncation attacks. The author's original publication [Har16a] was the first work in the literature to formally model these attacks as well as completeness.

We proposed a new scheme that can detect both omissions from excerpts as well as truncation attacks, and proven that our scheme satisfies our security notion.

Our scheme can be instantiated with an arbitrary forward-secure signature scheme, and can therefore be tuned to specific performance requirements and based on a wide variety of computational assumptions.

[15] The values shown here are an average per log entry.

5. Fault-Tolerant Sequential Aggregate Signatures for Robust Logging

This chapter introduces a cryptographic technique that facilitates a *trade-off* between two rivaling goals for the design of a cryptographic logging scheme: On the one hand, it is desirable for a cryptographic logging scheme to be able to identify unmodified log entries even if other log entries have been illicitly forged by an attacker. This functionality might be achieved by storing an individual signature for each log message, such as in [BY97; Hol06] and the SALVE scheme (see Chapter 4). On the other hand, a scheme with as little storage overhead as possible is preferable efficiency-wise. (For example, Ma and Tsudik [MT08; MT09] propose a scheme that only needs to save a single aggregate signature for an entire log file.) We achieve a trade-off between these goals by creating multiple (sequentially) aggregate signatures for different subsequences of the log records in the log file. The details of our construction are described below.

This chapter is strongly based on [Har+17b]. Significant parts of the former publication are reproduced here without or with only minor changes, and without specific designation. The publication [Har+17b] in turn applies a technique from [Har+16], which was independently developed by [Ida15; Ida+15].

5.1. Introduction

We consider the problem of creating a cryptographic scheme for securely storing log data that combines the following two properties:

- the logging scheme is *robust* to modifications, i.e. it can identify unmodified log entries as authentic, even if some other log entries have been illegitimately modified, and
- the logging scheme has sub-linear storage overhead in the number of stored log entries.

Previous solutions either stored one signature per log record, attaining robustness but having linear storage overhead (e.g. [BY97; Hol06], the "immutable" schemes in [MT09] and the SALVE scheme in Chapter 4), or stored a single signature for all log entries in the log file [MT08; MT09; Bul+14], attaining space-efficiency at the cost of being *fragile* (i.e. not being robust).[1]

[1] Buldas et al. [Bul+14] also realize this issue and propose a modification to their scheme which induces linear storage overhead. This modified scheme thus belongs to the former category.

We propose a trade-off between robustness and storage overhead, using a technique from [Har+16], independently developed by [Ida15, Chapter 4] and published in [Ida+15]. More concretely, we build a scheme where log records are redundantly authenticated by multiple (sequential) aggregate signatures. Each aggregate signature authenticates a subsequence of the complete log file. The subsequences are chosen in a specific way, as to guarantee that changes to up to d log records (where d is a parameter of our scheme) can be tolerated without affecting the scheme's ability to verify the authenticity and integrity of unmodified log entries.

More specifically, we use incidence matrices of *cover-free families* (see e.g. [KS64]) to determine the subsequences. Cover-free families are objects from the field of combinatorics, where a "base set" (or *universe*) $\mathcal{S}$ is divided into subsets (or *blocks*) B, such that the union of up to d blocks $B_1, \ldots, B_d$ does not "cover" any other block $B \notin \{B_1, \ldots, B_d\}$. We give a formal definition in Section 5.2.1.

We now briefly illustrate the key technique with an example from [Har+16]. Consider the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

We associate each column with a log entry and each row with an aggregate signature to be computed. The 1-entries in each row indicate the log entries to be included in the respective aggregate signature. Thus, if we had six log entries $m_1, \ldots, m_6$, the first sequential aggregate signature σ_1 would be computed from m_1, m_4, and m_6, another aggregate signature σ_2 would be based on m_1, m_2, and m_5, and so on.

Now suppose that an attacker modified the fourth entry of the log file m_4. As a result, verification of the first and third row would (most probably) fail. However, the second and fourth row are unaffected. One may thus verify the integrity and authenticity of m_1, m_2 and m_5 by checking the signature of the second row σ_2, and confirm the integrity and authenticity of m_3, m_5 and m_6 with the signature σ_4. Together, the signatures σ_2 and σ_4 authenticate the messages m_1, m_2, m_3, m_5 and m_6, so all unmodified log entries can still be verified. The reader may check that the latter property holds independently of which of the log records $m_1, \ldots, m_6$ is changed by the attacker, as long as only a single log entry is changed.[2] We will discuss how to instantiate our scheme such that it can accomodate more log entries, compensate more modifications and achieve better "compression" below.

The approach described above was originally developed in [Har+16] for (fully flexible) aggregate signatures. However, the technique equally lends itself to our context of detecting illicit modifications to log files. It realizes a black-box transformation from a forward-secure sequential aggregate signature scheme FSSAS to a *fault-tolerant* forward-secure sequential aggregate signature scheme. A signature of the fault-tolerant scheme

[2] Changing more than a single log entry may affect the verifiability of unchanged log entries. For example, changing m_4 and m_6 would lead to a situation where m_1, m_2 and m_5 can still be verified, but m_3 would be "lost". If an attacker changed m_4 and m_5 instead, all rows would be affected, and thus no log entry could be verified anymore.

is simply a "vector" of signatures $\sigma_1, \ldots, \sigma_r$ ($r \in \mathbb{N}$), each initialized to the empty signature λ of FSSAS. When a new message is added to the aggregate signature, the scheme assigns the next (leftmost) unused column to this message, determines the affected rows with the help of the matrix and uses the AggSign algorithm of FSSAS to update the signatures of the respective rows. Formally, the verification algorithm of a fault-tolerant scheme does not return a single bit indicating the validity of the entire claim sequence, but returns a sequence of bits instead, each indicating the validity of a specific claim.[3] We use the phrase "with list verification" to mark the syntactical difference, and then formally define the property of *fault-tolerance* for such schemes.

Note that our construction of fault-tolerance does not rely on any computational assumption, but fault-tolerance can be achieved unconditionally. (We only assume that a given set system is a CFF, which in turn can be shown unconditionally for certain constructions.) This does not mean we achieve unconditional "security", though, since the property of fault-tolerance is not related to "security" of signature schemes (in the sense of unforgeability notions such as the ones in Sections 2.3.1 and 2.3.2), but is a generalization of *correctness*. While *fault-tolerance* can be shown unconditionally, we still require computational assumptions for the *security* of our schemes.

We give another transformation, building a robust logging scheme from a fault-tolerant forward-secure SAS scheme and a (plain) forward-secure signature scheme. This transformation essentially adds security against reordering and deletion of claims, including security against truncations to the fault-tolerant signature scheme. We use the technique discussed in Section 4.3 to achieve truncation security.

Having a fixed matrix as in the example above places a limit on the number of messages which can be signed by our scheme. However, since the number of messages which can be added can be made exponentially large with only polynomial overhead, this does not pose a serious problem. Nonetheless, we briefly discuss how to achieve a scheme without such a limitation in Section 5.3.4.

The technique from [Har+16] also features so-called selective verification: To verify a single log entry, one can use the aggregate signature's redundancy to call the verification routine on a smaller set, instead of the whole log file, see Section 5.3.4. Note that space-inefficient logs using distinct signatures for each log entry have this feature trivially.

Our Contribution.

We build a cryptographic logging scheme that simultaneously features robustness, sublinear storage overhead and truncation security. Our approach is provably secure and uses a tight security reduction. For this, we define a security model for the logging scenario that captures truncation attacks as well as a wide range of other manipulations. This distinguishes our work from previous publications (e.g. [MT07a; Ma08]) where

[3] In the definitions given in the original publications [Har+16; Har+17b], the verification algorithm returns a (multi-)set of all valid claims. We redefine the verification algorithm to output a sequence of bits instead, since it seems more natural in retrospect.

truncation security is only argued for informally and is not part of the security model, or which allow for omission of log entries [YPR12b].

Our logging scheme can operate in the stand-alone model without any interaction with another party. It does not require public ledgers (e.g. blockchains) or any other third party that needs to vouch for the integrity of the log file. However, our scheme can easily be combined with such techniques, and thus can be re-used as a building block for future schemes.

We implemented a prototype version of our scheme and present benchmark results for multiple sets of parameters.

Outline.

Section 5.2 introduces notation, preliminary definitions and tools we need for our constructions. In Section 5.3 we present the notion of *fault-tolerant* forward-secure sequential aggregate signatures, and give a generic construction of such schemes from an arbitrary forward-secure sequential aggregate signature scheme. We then apply the technique from Section 4.3 to this scheme in order to add truncation resistance and obtain a robust and truncation-secure logging scheme, see Section 5.4. We present an example implementation of this scheme as well as benchmark results in Section 5.5. Finally, we conclude in Section 5.6.

5.2. Preliminaries

We briefly introduce some additional preliminaries for this chapter.

5.2.1. Cover-Free Families

Definition 5.1 (Cover-free Family, based on [KS64; KRS99])**.** Let $\mathcal{S}$ be a finite set, $\mathcal{B}$ be a set of subsets (or *blocks*) of $\mathcal{S}$ and $d \in \mathbb{N}$. The pair $\mathcal{F} = (\mathcal{S}, \mathcal{B})$ is a *d-cover-free family* (or *d-CFF*) if for all d blocks $B_1, \ldots, B_d \in \mathcal{B}$ and all distinct $B \in \mathcal{B} \setminus \{B_1, \ldots, B_d\}$, we have that $B \not\subseteq B_1 \cup \cdots \cup B_d$, i.e. no block is covered by the union of any other d blocks. $\mathcal{F}$ is a *cover-free family (CFF)* if it is d-cover-free for a $d \geq 1$. A CFF with a linear order $\leq$ on $\mathcal{B}$ is called *ordered*.

To simplify the presentation, we also assume an order on $\mathcal{S}$ and usually identify $\mathcal{S}$ with $[r]$, for $r = |\mathcal{S}|$.

Definition 5.2 (Incidence Matrices)**.** The *incidence matrix* $\mathcal{M}$ of an ordered CFF $(\mathcal{S}, \mathcal{B})$ is defined via

$$\mathcal{M}[i,j] = \begin{cases} 1, & \text{if } i \in B_j, \\ 0, & \text{otherwise,} \end{cases}$$

for $|\mathcal{S}| = r \in \mathbb{N}$, $|\mathcal{B}| = n \in \mathbb{N}$, $i \in [r]$, $\mathcal{B} = \{B_1 \leq \cdots \leq B_n\}$ and $j \in [n]$.

In this way, each row of the incidence matrix is associated with an element $i \in \mathcal{S}$, and the positions of the 1-entries in the row show the blocks B_j that contain i. Similarly, each column of the incidence matrix corresponds to a block B_j, and the 1-entries designate the elements $i \in \mathcal{S}$ contained in B_j.

While our construction outlined below can be instantiated with an arbitrary cover-free family, we briefly present the construction from [KRS99, Section 5] as an example of a CFF in Appendix A.

5.2.2. Notation

If $\mathcal{M}$ is a matrix with n rows and m columns and $i \in [n]$, $j \in [m]$, then $\mathcal{M}[i, 1 \dots j]$ is the sequence of the leftmost j entries of the i-th row of $\mathcal{M}$. Moreover, if S is a sequence over $\{0, 1\}$, then $\mathrm{ones}(S)$ is the sequence of all $k \in [\mathrm{len}(S)]$ such that $S[k] = 1$. (This sequence is sorted in ascending order.) Consequently, if $\mathcal{M}$ is a matrix over $\{0, 1\}$, then $\mathrm{ones}(\mathcal{M}[i, 1 \dots j])$ is the sequence of all $k \in [j]$ such that $\mathcal{M}[i, k] = 1$.

During this chapter, we will frequently use $\mathrm{ones}(\mathcal{M}[i, 1 \dots j])$ as an index sequence (as defined in Definition 2.14 on page 23) for a claim sequence $C = (c_k)_{k=1}^{l}$ $(l \in \mathbb{N}, l \geq j)$. The subsequence of C induced by $\mathrm{ones}(\mathcal{M}[i, 1 \dots j])$ is $C[\mathrm{ones}(\mathcal{M}[i, 1 \dots j])]$, i.e. the subsequence of C that contains all c_k such that $\mathcal{M}[i, k] = 1$. In order to simplify notation, we denote this subsequence by $C[\mathcal{M}, i, 1 \dots j]$.

5.3. Fault-Tolerant Forward-Secure Sequential Aggregate Signatures

In this section we define the syntax of forward-secure sequential aggregate signatures (SAS) with list verification, define fault-tolerance for such schemes, and present a security notion which captures the forward security property. We then give a generic construction, and prove its fault-tolerance and security.

We now define key-evolving sequential aggregate signature schemes (SAS) with list verification. The definition given below is mostly the same as the definition of (standard) key-evolving sequential aggregate signature schemes (see Definition 2.22 on page 32), but differs in the verification algorithm and the definition of validity.

Definition 5.3 (Key-Evolving Sequential Aggregate Signature Scheme with List-Verification)**.** A *key-evolving SAS scheme with list-verification* Σ is a tuple of four PPT algorithms $\Sigma = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$, where:

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk})$
: takes as input the security parameter κ and an upper bound T on the number of epochs. It outputs a key pair $(\mathsf{sk}_1, \mathsf{pk})$, where sk_1 is the secret key for the first epoch.

$\mathsf{Update}(\mathsf{sk}_t) \to \mathsf{sk}_{t+1}$
: takes as input the secret key sk_t of period t. If $t \geq T$ the output of Update is not

defined. If $t < T$ it computes the secret key sk_{t+1} for period $t+1$ and securely and irrecoverably erases the old key sk_t. It outputs sk_{t+1}.

$\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i) \to \tau_i$
takes as input a secret key sk_t for an epoch t, a claim sequence C_{i-1}, a corresponding signature τ_{i-1} and a message m_i. It outputs a signature τ_i for the new claim sequence $C_i := C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$.

$\mathsf{Verify}(C, \tau) \to V$
takes as input a claim sequence C of length $l \in \mathbb{N}_0$ and a signature τ for C and outputs a sequence $V \in \{0,1\}^l$ of l bits. Claims $C[i]$ for which $V[i] = 1$ are taken to be valid ($i \in [l]$), those claims with $V[i] = 0$ are considered invalid.

We require a key-evolving sequential aggregate signature scheme with list verification to be correct as defined below.

Definition 5.4 (Valid and Regular Signatures)**.** Let Σ be a quadruple of algorithms as defined above, C_i be a claim sequence and τ_i be a signature. We say that τ_i is *valid for position j of C_i* (where $j \in [\mathrm{len}(C_i)]$), if $\mathsf{Verify}(C_i, \tau_i)[j] = 1$.

We say that τ_i is *regular* for C_i iff either

- $C_i = ()$ and $\tau_i = \lambda$, or
- $C_i = C_{i-1} \,\|\, c$ for some claim $c = (\mathsf{pk}, t, m_i)$ with $t \in [T]$, and τ_i is in the image of $\mathsf{AggSign}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i)$ where τ_{i-1} is a regular signature for C_{i-1}, m_i is an arbitrary message, $T = T(\kappa) \in \mathrm{poly}(\kappa)$, and sk_t is the $(t-1)$-times updated version of some secret key sk_1 such that $(\mathsf{sk}_1, \mathsf{pk})$ is a key-pair output by $\mathsf{KeyGen}(1^\kappa, 1^T)$.

5.3.1. Fault Tolerance of FS-SAS Schemes

Definition 5.5 (Differences of Claim Sequences)**.** Let $n, n' \in \mathbb{N}_0$, and $C = (c_1, \ldots, c_n)$, $C' = (c'_1, \ldots, c'_{n'})$ be claim sequences. We say that C and C' *differ on ℓ positions* ($0 \le \ell \le \min(n, n')$) iff $c_i \neq c'_i$ for ℓ indices $1 \le i \le \min(n, n')$ and $c_i = c'_i$ for the rest. Moreover, we say that C' *contains d errors* with respect to C iff they differ on ℓ positions and $d = |n - n'| + \ell$.

Definition 5.6 (Fault Tolerance)**.** A key-evolving SAS scheme Σ with list verification is *tolerant against d errors*, iff for all claim sequences $C = (c_1, \ldots, c_n)$, $C' = (c'_1, \ldots, c'_{n'})$ ($n, n' \in \mathbb{N}_0$) where C' contains at most d errors with respect to C and for all signatures τ which are regular for C, we have

$$V[i] = 1 \quad \text{for all } 1 \le i \le \min(n, n') \text{ where } c_i = c'_i,$$

where $V := \Sigma.\mathsf{Verify}(C', \tau)$. In other words, Verify correctly identifies *at least* all claims c_i from C' as valid that were already contained in C at the same position.

A *d-fault-tolerant key-evolving SAS scheme* is an SAS scheme with list verification that is tolerant against d errors. A scheme is *fault-tolerant*, if it is d-fault-tolerant for some $d > 0$.

Note that $\Sigma.\mathsf{Verify}$ may also identify claims c as valid where $c = C'[i] \neq C[i]$, but our security proof will show that such events are extremely rare or trivial.

Observe that 0-fault-tolerance requires that if $C = C'$ and the signature τ is regular (for C) then $V[i] = 1$ for all claims $c_i \in C$. This observation illustrates that fault-tolerance is not a property akin to "security" (in the sense of unforgeability notions), but a generalization of correctness. Hence we may define correctness as a special case of fault-tolerance:

Definition 5.7 (Correctness)**.** We say that a key-evolving SAS scheme with list verification is *correct* iff it is *0-fault-tolerant*.

Note that our definition of key-evolving sequential aggregate signature schemes *with list verification* represents a purely syntactical change. To show this, we build a simple example scheme with list verification (but without fault-tolerance) from a scheme without list verification.

Example 5.8 (A Scheme with List Verification but without Fault Tolerance)**.** Let FSSAS be a key-evolving sequential aggregate signature scheme (without list verification). We define a scheme FSSAS' *with* list verification as follows: The key generation, key evolution and signing algorithms of FSSAS' are the same as for FSSAS. The verification algorithm of FSSAS' first runs the verification algorithm of FSSAS on its inputs C, σ and obtains a single bit b indicating the validity of all claims in C. If $b = 1$, then it outputs the sequence $1^{\mathrm{len}(C)}$, otherwise it outputs $0^{\mathrm{len}(C)}$.

The resulting scheme is correct and secure (as defined below) if FSSAS is correct and secure, respectively. However, the resulting scheme is not fault-tolerant: A single invalid signature will still lead to a situation where no original message can be verified anymore. Thus, the scheme still exhibits an "all-or-nothing" behavior during verification, instead of the desired "graceful degradation".

5.3.2. Security Notion

Let $\mathsf{AS} = (\mathsf{KeyGen}, \mathsf{Update}, \mathsf{AggSign}, \mathsf{Verify})$ be a key-evolving SAS scheme with list verification, $\mathcal{A}$ be a PPT algorithm, $\kappa \in \mathbb{N}$ a security parameter, and T be the number of epochs. The security experiment for a forward-secure SAS scheme with list verification is identical to that of forward-secure SAS schemes described in Section 2.3.2, except for the following difference: The experiment outputs 1 iff there is a non-trivial claim $c^* = C^*[i]$ (for $i \in [\mathrm{len}(C^*)]$) such that $V[i] = 1$, where $V := \mathsf{Verify}(C^*, \sigma^*)$.

A key-evolving SAS scheme with list verification is called *forward-secure sequential aggregate signature existentially unforgeable under chosen message attacks* (*FS--SAS-EUF-CMA-secure*) if for all $T = T(\kappa) \in \mathrm{poly}(\kappa)$, the probability of each PPT adversary $\mathcal{A}$ to win the above experiment is negligible in κ.

5.3.3. Generic Construction

Next, we show how to port the generic construction of fault-tolerant aggregate signatures given by [Har+16] to forward-secure sequential aggregate signature schemes. We use

this construction to convert a forward-secure SAS scheme FSSAS to a fault-tolerant forward-secure SAS scheme. We give proofs of both security and fault-tolerance below.

Let FSSAS be a key-evolving SAS scheme, $\mathcal{F}$ a d-cover-free family ($d \in \mathbb{N}_0$), and $\mathcal{M}$ its incidence matrix. A signature τ of our new scheme is a vector of signatures σ of FSSAS. The algorithms of our scheme are as follows:

KeyGen and Update
: are identical to the respective algorithms of FSSAS.

$\mathsf{AggSign}(\mathsf{sk}_t, C_{j-1}, \tau_{j-1}, m_j) \to \tau_j$
: takes as input a secret key sk_t, a claim sequence $C_{j-1} = (c_1, \ldots, c_{j-1})$, its corresponding signature τ_{j-1} and a message m_j to sign. The sequential aggregate signature is updated component-wise, according to the entries of $\mathcal{M}$. More precisely, we set

$$\tau_j[i] \coloneqq \mathsf{FSSAS.AggSign}(\mathsf{sk}_t, C_{j-1}[\mathcal{M}, i, 1 \ldots j-1], \tau_{j-1}[i], m_j),$$

where $\mathcal{M}[i, j] = 1$, and let $\tau_j[i] \coloneqq \tau_{j-1}[i]$ otherwise ($i \in [\mathrm{rows}(\mathcal{M})]$). (Here, $C_0 \coloneqq ()$ and $\tau_0[i] \coloneqq \lambda$ for each i.) The output is τ_j.

$\mathsf{Verify}(C, \tau) \to V$
: takes as input a claim sequence C of length $n \in \mathbb{N}_0$ and an aggregate signature τ for C. We compute a bit sequence $V \in \{0,1\}^n$ that specifies for each claim if it is considered valid. We initialize V to 0^n, and iterate over all entries $\tau[i]$ of τ. In each iteration, if $\mathsf{FSSAS.Verify}(C[\mathcal{M}, i, 1 \ldots n], \tau[i]) = 1$ we let $V \coloneqq V \vee \mathcal{M}[i, 1 \ldots n]$. (Here, $\vee$ denotes the bitwise logical OR of two bit strings.) Finally, output V.

This concludes the description of our scheme. Before proving the scheme fault-tolerant and secure, we first show the following lemma, which shows the main invariant maintained by our scheme:

Lemma 5.9. Let τ be a regular signature of the scheme described above for a claim sequence C of length $n \in \mathbb{N}_0$. Then, for all $i \in [r]$, $\tau[i]$ is regular for $C[\mathcal{M}, i, 1 \ldots n]$, where $r = |\mathcal{S}|$.

Proof. We show the lemma by a straightforward induction over n. For the start of the induction, let $n = 0$. Then $C = ()$ and hence $C[\mathcal{M}, i, 1 \ldots n] = ()$ for all i. Moreover, for all i, $\tau[i] = \lambda$, which is regular for $() = C[\mathcal{M}, i, 1 \ldots n]$. This constitutes the start of the induction.

For the induction step, assume that for a fixed $n' \in \mathbb{N}_0$, all claim sequences C' of length n', all signatures τ' regular for C' and all $i \in [r]$, it holds that $\tau'[i]$ is regular for $C'[\mathcal{M}, i, 1 \ldots n']$.

Let C be a claim sequence of length $n = n' + 1$, $i \in [r]$, and τ be regular for C. Let C' be the length-n' prefix of C and $c = C[n]$. Since τ is regular for C, τ is in the image of $\mathsf{AggSign}(\mathsf{sk}_t, C', \tau', m)$ for some secret key sk_t, $t \in [T]$, some message $m \in \{0,1\}^*$, and τ' which is regular for C'. By the induction hypothesis, we thus know that for all $i \in [r]$, we have that $\tau'[i]$ is regular for $C'[\mathcal{M}, i, 1 \ldots n']$.

Now, if $\mathcal{M}[i,n] = 0$, then $\tau[i] = \tau'[i]$, which is regular for $C'[\mathcal{M}, i, 1 \dots n'] = C[\mathcal{M}, i, 1 \dots n]$. If $\mathcal{M}[i,n] = 1$ instead, then (by the construction of $\mathsf{AggSign}$), we have that $\tau[i]$ is in the image of $\mathsf{AggSign}(\mathsf{sk}_t, C'[\mathcal{M}, i, 1 \dots n'], \tau'[i], m)$, and hence it is regular for $C' \parallel c = C[\mathcal{M}, i, 1 \dots n]$.

So in both cases ($\mathcal{M}[i,n] = 0$ and $\mathcal{M}[i,n] = 1$) we have that $\tau[i]$ is regular for $C[\mathcal{M}, i, 1 \dots n]$, as claimed. □

We now prove the scheme fault-tolerant and secure.

Theorem 5.10. Let Σ be the key-evolving SAS scheme with list verification defined above. If Σ is based on a d-CFF $\mathcal{F} = (\mathcal{S}, \mathcal{B})$, and FSSAS is correct, then Σ is tolerant against d errors.

We give a sketch of the proof first. Observe that each message m_j is redundantly aggregated into several of the signatures $\tau[i]$, namely those where $\mathcal{M}[i,j] = 1$. Each column j is associated with a block B_j of $\mathcal{F}$. If errors occur on (at most) d positions $(k_1, \dots, k_d)$, verification of the rows $S_K = B_{k_1} \cup \dots \cup B_{k_d}$ will most likely fail, whereas the other rows are unaffected. However, this subset S_K cannot cover any other block B_j belonging to another message m_j due to the cover-freeness of $\mathcal{F}$. Thus, each correct message m_j can be verified from at least one row i, and the verification algorithm will set $V[j]$ to 1 in the i-th iteration. Thus, the message will be considered valid by our scheme. We now proceed to the formal argument, which is illustrated by Figure 5.1.

Proof. Let C and C' be two claim sequences, where C' contains at most d errors with respect to C. Let n, n' be the lengths of C, C', respectively, let τ be a regular signature for C, and $\mathcal{M}$ be the incidence matrix of $\mathcal{F}$, let $\mathcal{S} = \{s_1, \dots, s_r\}$, $\mathcal{B} = \{B_1, \dots, B_m\}$ and $r = \mathrm{rows}(\mathcal{M}) = |\mathcal{S}|$. Let $V \coloneqq \mathsf{FSSAS.Verify}(C', \tau)$.

We need to show that $V[j] = 1$ for all $1 \le j \le \min(n, n')$ where $C[j] = C'[j]$. Observe that V is the bitwise logical OR of all sequences $\mathcal{M}[i, 1 \dots n']$ where

$$\mathsf{FSSAS.Verify}(C'[\mathcal{M}, i, 1 \dots n'], \tau[i]) = 1. \tag{5.1}$$

Therefore, $V[j]$ will be 1 if for at least one of these i it holds that $\mathcal{M}[i,j] = 1$. So, let $j \in [\min(n, n')]$ be an arbitrary position where $C[j] = C'[j]$. We will show that there is at least one row i such that (5.1) holds and $\mathcal{M}[i,j] = 1$.

Let $D \coloneqq \{k \in [\min(n, n')] \colon C[k] \neq C'[k]\}$ be the indices of all positions where C and C' differ, and let $E \coloneqq \{\min(n, n') + 1, \dots, \max(n, n')\}$ be the set of indices of excess claims. Let $K \coloneqq D \cup E$. Since C' contains at most d errors with respect to C, we have $|K| \le d$. Clearly, $j \notin K$, because j is a position where C and C' agree, whereas K contains the positions k where either they disagree, or one of the claim sequences does not even have k elements.

Define $I_K \coloneqq \{i \in [r] \colon \mathcal{M}[i,k] = 1 \text{ for a } k \in K\}$. This is precisely the set of rows such that $C[\mathcal{M}, i, 1 \dots n] \neq C'[\mathcal{M}, i, 1 \dots n']$, i.e. the rows affected by the errors. For all other rows $i \notin I_K$ it holds that $C[\mathcal{M}, i, 1 \dots n] = C'[\mathcal{M}, i, 1 \dots n']$.

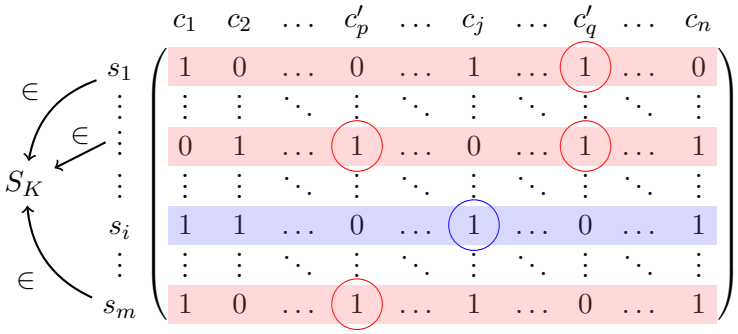

Figure 5.1.: An example for our proof of fault tolerance. Recall that we associate elements $s \in \mathcal{S}$ with rows of the incidence matrix. Likewise, we associate claims with blocks of the CFF, and thus with columns of the matrix. Assume that the incidence matrix belongs to a d-CFF with $d \geq 2$. Let $C = (c_i)_{i=1}^n$ and C' be claim sequences, differing on positions $K = \{p, q\}$, and w.l.o.g. $p < q$. The ones in the respective columns (in red circles) define a subset S_K of the rows of the matrix (red background). I_K is the set of indices of these rows. Since S_K is the union of B_p and B_q, and the matrix belongs to a d-CFF with $d \geq 2$, S_K does not cover B_j, so there is a $s_i \in B_j$ which is not contained in S_K. The fact that s_i belongs to B_j is indicated by a one in a blue circle. Thus, verification of row i (blue background) will succeed, and hence all claims belonging to this row (and in particular c_j) can be successfully verified. Note that c_j could in theory also be verified from the first or the last row, but these rows are affected by the changes at the indices p and q of the claim sequence.

Let $S_K \coloneqq \bigcup_{k \in K} B_k$. Note that

$$\begin{aligned} S_K &= \{s_i \in \mathcal{S} \colon s_i \in B_k \text{ for a } k \in K\} \\ &= \{s_i \in \mathcal{S} \colon \mathcal{M}[i,k] = 1 \text{ for a } k \in K\} \\ &= \{s_i \in \mathcal{S} \colon i \in I_K\}, \end{aligned}$$

where the first equality follows from the definition of S_K and the definition of union of sets, the second equality follows directly from the definition of the incidence matrix $\mathcal{M}$, and the third equality follows from the definition of I_K.

As a union of at most d blocks $B_k \in \mathcal{B}$, S_K does not cover any other, distinct block $B \in \mathcal{B}$. In particular, since $j \notin K$, S_k does not cover B_j. So, there must be a $s_i \in B_j$ such that $s_i \notin S_K$, and thus $i \notin I_K$. At the same time, we know $\mathcal{M}[i,j] = 1$, since $s_i \in B_j$.

All that is left to show is that (5.1) holds for this i. However, since $i \notin I_K$, we have $C'[\mathcal{M}, i, 1 \dots n'] = C[\mathcal{M}, i, 1 \dots n]$, and thus we may replace $C'[\mathcal{M}, i, 1 \dots n']$ by $C[\mathcal{M}, i, 1 \dots n]$ in (5.1). Since τ is regular for C, we have that $\tau[i]$ is regular for $C[\mathcal{M}, i, 1 \dots n]$ by Lemma 5.9. Therefore (5.1) must hold due to the correctness of FSSAS. □

Next, we will prove the security of our generic construction given above.

Theorem 5.11. Let FSSAS be a key-evolving SAS scheme, $\mathcal{F}$ be a cover-free family with incidence matrix $\mathcal{M}$, and Σ be the scheme described above. If there exists a PPT algorithm $\mathcal{A}$ that breaks the security of Σ with success probability $\varepsilon_{\mathcal{A}}$, then there exists a PPT attacker $\mathcal{B}$ that breaks the FS-SAS-EUF-CMA-security of FSSAS with success probability $\varepsilon_{\mathcal{B}} \geq \varepsilon_{\mathcal{A}}$.

Again, we first give a proof sketch before moving to the formal proof. Note that the verification algorithm of our scheme outputs the logical OR of all $\mathcal{M}[i, 1 \dots n]$ for all valid rows i. Thus, to break the security, the attacker must create a forgery where a non-trivial claim is contained in a valid row, which constitutes a successful attack on the underlying scheme FSSAS.

Proof. The proof is mostly analogous to the proof from [Har+16]. Define $\mathcal{B}$ as follows. $\mathcal{B}$ internally simulates the security experiment for forward-secure SAS schemes with list verification for $\mathcal{A}$. $\mathcal{B}$'s input consists of a public key pk^* and the number of epochs 1^T. $\mathcal{B}$ forwards these to $\mathcal{A}$. In the following, we describe how $\mathcal{B}$ responds to $\mathcal{A}$'s oracle queries.

Signature Oracle.

When $\mathcal{A}$ queries its signature oracle with a claim sequence C_{j-1}, a signature τ_{j-1} and a message m_j, then $\mathcal{B}$ queries its signature oracle multiple times, one time for each row i of $\mathcal{M}$ with $\mathcal{M}[i,j] = 1$. For each such i, $\mathcal{B}$ queries its signature oracle with inputs $C_{j-1}[\mathcal{M}, i, 1 \dots j-1]$, $\tau_{j-1}[i]$ and m_j and obtains $\tau_j[i]$. For all remaining $i \in \text{rows}(\mathcal{M})$, $\mathcal{B}$ sets $\tau_j[i] \coloneqq \tau_{j-1}[i]$. $\mathcal{B}$ returns the resulting signature τ_j to $\mathcal{A}$.

Epoch Switching Oracle.
When $\mathcal{A}$ queries its epoch switching oracle, $\mathcal{B}$ does the same and returns the string "ok" to $\mathcal{A}$.

Break-In Phase.
If $\mathcal{A}$ decides to break in, $\mathcal{B}$ does the same in the FS-SAS-EUF-CMA experiment, and thereby obtains the secret key sk_t, where t is the current epoch. $\mathcal{B}$ then returns sk_t to $\mathcal{A}$.

Finally, $\mathcal{A}$ outputs a claim sequence C^* of length n^* and a signature τ^*. If $\mathcal{A}$ does not win the experiment, then $\mathcal{B}$ aborts. For the remainder of this proof, assume that $\mathcal{A}$ wins the experiment. Then (by definition) C^* contains a non-trivial claim $c^* = (\mathsf{pk}^*, t^*, m^*) = C^*[j^*]$ (for a $j^* \in [n^*]$) such that $V[j^*] = 1$, where $V \coloneqq \Sigma.\mathsf{Verify}(C^*, \tau^*)$.

Since $V[j^*] = 1$ and V is the logical OR of all $\mathcal{M}[i, 1 \dots n^*]$ where

$$\mathsf{FSSAS}.\mathsf{Verify}(C^*[\mathcal{M}, i, 1 \dots n^*], \tau^*[i]) = 1, \tag{5.2}$$

there must be an index i such that $\mathcal{M}[i, j^*] = 1$ and (5.2) holds. $\mathcal{B}$ searches for such a claim c^* and a row index i. $\mathcal{B}$ then outputs $C^*[\mathcal{M}, i, 1 \dots n^*]$ and $\tau^*[i]$ as forgery for FSSAS. This concludes the description of the attacker $\mathcal{B}$.

Non-Triviality. We now argue that $\mathcal{B}$'s output is non-trivial. In each epoch, $\mathcal{B}$ outputs exactly the same messages to its signature oracle that $\mathcal{A}$ outputs. Therefore, if $\mathcal{A}$ did not output m^* to its signature oracle during epoch t^*, then $\mathcal{B}$ did not output m^* to its signature oracle during epoch t^* either, and so the claim c^* is non-trivial for $\mathcal{B}$, too. Moreover, we have $\mathcal{M}[i, j^*] = 1$, so c^* is contained in $\mathcal{B}$'s output $C^*[\mathcal{M}, i, 1 \dots n^*]$.

Validity. (5.2) holds by the choice of i.

Summary. We now argue that $\varepsilon_{\mathcal{B}} \geq \varepsilon_{\mathcal{A}}$. First observe that $\mathcal{B}$ perfectly simulates the FS-SAS-EUF-CMA experiment for $\mathcal{A}$. If $\mathcal{A}$ wins, then $\mathcal{B}$'s output is valid and contains a non-trivial claim c^*, hence $\mathcal{B}$ wins the security experiment for FSSAS. Thus, $\varepsilon_{\mathcal{B}} \geq \varepsilon_{\mathcal{A}}$. Furthermore, $\mathcal{B}$ runs in polynomial time if $\mathcal{A}$ does. □

Corollary 5.12. If FSSAS is FS-SAS-EUF-CMA-secure then Σ is FS-SAS-EUF-CMA-secure.

5.3.4. Discussion

We now review our scheme and discuss some of its properties as well as possible extensions.

Selective Verification.

Let C be a claim sequence, $n := \text{len}(C)$, and τ be a signature for C. Suppose one is interested in verifying a specific claim $c = C[j]$ ($j \in [n]$), but not the claim sequence as a whole. To achieves this, one might simply run Verify on C, and check if the output at position j is 1. However, our scheme features a more efficient way of checking individual claims.

Observe that the output of the Verify algorithm is the bitwise logical OR of all rows i of the incidence matrix $\mathcal{M}$ where $\mathsf{FSSAS.Verify}(C[\mathcal{M}, i, 1 \dots n], \tau[i]) = 1$. Thus, there is no need to check rows i where $\mathcal{M}[i, j] = 0$, since these rows are irrelevant to the outcome for the claim c. Moreover, once a row with a valid signature $\tau[i]$ and $\mathcal{M}[i, j] = 1$ has been found, there is no more need to check the other rows, since $V[i]$ has been set to 1, and adding further rows by bitwise logical OR will not change the outcome for claim c. Thus, it may be possible to verify the claim c by just checking a single row of $\mathcal{M}$.

Unbounded CFFs.

Note that our scheme described above cannot aggregate an unbounded number of signatures: Each message is associated to one of the blocks of the cover-free family (and hence, to one of the columns of the CFF's incidence matrix). Thus, once a CFF has been fixed, the CFF limits the number of signatures (and hence messages) which can be aggregated. We now discuss how to deal with this issue.

Firstly, one might accept this limitation, provided that the number of messages which can be aggregated is "large enough". For example, when fixing the parameter d, the number of blocks of the CFF given in Appendix A grows *exponentially* in the size of the CFF's universe (see [Har+16]). Hence, it is possible to aggregate signatures for an exponential number of messages, while only needing to store a polynomial number of (aggregate) signatures. Rephrasing this, it is possible to "compress" signatures for n messages to $\mathcal{O}(\log n)$ aggregate signatures while maintaining fault-tolerance. Table A.1 (see page 140) gives some example parameters for CFFs supporting at least one billion, 2^{40}, 2^{50} and 2^{64} blocks.

[Har+16] showed that this logaritmic behaviour is asymptotically optimal: It is necessary for a fault-tolerant signature to have at least $\Omega(\log n)$ bits, where n is the number of blocks of the CFF. This follows from an information-theoretical argument, see [Har+16] for details.

However, if one prefers to have a logging scheme which can support a truly unbounded number of messages, then a different solution is needed. [Har+16] proposed to use "unbounded" CFFs, which can be "enlarged" once the available blocks are exhausted. However, the simple construction of unbounded CFFs given by [Har+16] has a *linear* relationship between the size of the CFF's universe and the number of blocks, instead of the desired logarithmic one. Idalino and Moura [IM18] give for more efficient constructions of unbounded CFFs.

Generalization to More Types of Changes.

Recall Definition 5.5: We say that a claim sequence C' contains d errors with respect to a claim sequence C iff they differ on l positions (for an $l \in \mathbb{N}_0$) and their lengths $n := \text{len}(C)$, $n' := \text{len}(C')$ differ by $d - l$. According to this definition, the following types of modifications to C are counted as a single error:

- changing or replacing one claim in C,
- appending a new claim to C, and
- removing the last claim of C.

In contrast, if an attacker deleted some claim c_i for some $i < \text{len}(C)$ and the trailing claims are "moved forward", then (according to this definition) the resulting claim sequence C' had $n - i + 1$ errors with respect to C. Similarly, if an attacker inserted a new claim between positions $i - 1$ and i, such that the new claim is at position i and all trailing claims are moved towards higher indices, then C' had $n - i + 2$ errors with respect to C. This behavior stands in contrast to our expectations, if we consider these types of modifications as just a *single* change. We now briefly outline an idea that might be followed towards solving this problem.

One might consider not assigning the columns to the messages in a left-to-right fashion, but choosing the column by hashing the respective message. For example, given a hash function $H : \{0,1\}^* \to [n]$ and a CFF's incidence matrix $\mathcal{M}$ with at least n columns, one might assign a message m to the column $H(m)$. We now discuss some effects that would be caused by this modification to our scheme.

Hash Collisions. Naturally, this change affects fault-tolerance if hash collisions occur: If two messages m_1, m_2 are hashed to the same column, then modification of a single one of them (say, m_1) will prevent the verification of the second one (m_2) since the rows from which m_2 can be verified are the same as the rows for m_1, and all of them are affected by the change to m_1.

We identify two options to deal with this issue: Firstly, one might simply accept this situation. In an application where messages are not chosen maliciously, a universal hash function together with a CFF having enough blocks may be a satisfactory solution. In this case, one would need to define a probabilistic notion of fault-tolerance.

If messages *are* chosen maliciously, however, one should consider our second option: using a collision-resistant hash function instead of a universal one. This guarantees that no PPT attacker has a non-negligible chance of finding two messages hashing to the same column index, and thus one would require a computational version of our definition of fault-tolerance. Moreover, this requires a hash function with a very large output space, and thus a CFF with a very large number of blocks. Table A.1 (see page 140) shows some instantiations of the CFF given in Appendix A where the number of blocks is slightly greater than 2^{256}, such that the CFF could be used with practical hash functions such as SHA-256 and SHA3-256.

Reduction of Fault Tolerance. Another effect caused by the hashing approach is that in order to tolerate d changes, we expect that one would require a $2d$-CFF (instead of a d-CFF).

Example 5.13. Consider the 1-CFF defined by the matrix given in Section 5.1. Let $C = (c_1, c_2) = ((\mathsf{pk}, t_1, m_1), (\mathsf{pk}, t_2, m_2))$ be a claim sequence, where $H(m_1) = 2$ and $H(m_2) = 4$. A regular signature τ for C would be generated as:

$$\tau \coloneqq \begin{pmatrix} \mathsf{AggSign}(\mathsf{sk}_{t_2}, (), \lambda, m_2) \\ \mathsf{AggSign}(\mathsf{sk}_{t_1}, (), \lambda, m_1) \\ \mathsf{AggSign}(\mathsf{sk}_{t_2}, (c_1), \mathsf{AggSign}(\mathsf{sk}_{t_1}, (), \lambda, m_1), m_2) \\ \lambda \end{pmatrix}$$

Now let $C' = (c_1, c_2')$, where $c_2' = (\mathsf{pk}, t_2, m_2')$ for $m_2' \neq m_2$, and $H(m_2') = 1$. Now $\mathsf{Verify}(C', \tau)$ would check if

$$\begin{array}{lll} \tau[1] & \text{is valid for} & (c_2'), \\ \tau[2] & \text{is valid for} & (c_1, c_2'), \\ \tau[3] & \text{is valid for} & (c_1), \quad \text{and} \\ \tau[4] & \text{is valid for} & (). \end{array}$$

Thus, we expect the verification on the first three rows to fail, and thus expect that Verify would output $(0, 0)$. Hence, even though C' only has a single error with respect to C, the unmodified claim c_1 can no longer be verified. We attribute this change to the fact that when c_2 was replaced by c_2', this affected *two* columns: both the column $H(m_2)$ *and* the column $H(m_2')$. This leads to the conjecture that for the variant of our scheme discussed here, a $2d$-CFF is needed in order to tolerate d changes.

Future Work. Formally examining this approach is left as future work. Moreover, finding a different approach towards building a fault-tolerant scheme which better tolerates addition and removal of claims at arbitrary positions is another interesting open problem.

5.4. Robust Secure Logging

We now focus on building a robust logging scheme with sub-linear storage overhead. Our construction makes use of an arbitrary fault-tolerant forward-secure SAS scheme and a plain forward-secure signature scheme. We use the approach highlighted in Section 4.3 to obtain truncation-security.

The syntax of our logging scheme is similar to the syntax of FT-FS-SAS schemes. The key difference is that the function of the verification algorithm is distributed over two algorithms: The first algorithm (called $\mathsf{VerifyLog}$) checks the overall log file and outputs either 1 or 0. Secondly, there is a $\mathsf{ValidEntries}$ algorithm which outputs a bit vector V indicating which of the log entries in the log file have a valid signature.

This algorithm behaves very similar to the verification algorithm of a fault-tolerant forward-secure SAS scheme. It should be called if the VerifyLog algorithm detects an error, in order to obtain more fine-grained information about which log entries are valid.

Note that even if the signature is valid in the sense that all claims are returned by ValidEntries, an attacker might have truncated the log. In this case VerifyLog returning 0 points towards this possibility.

For simplicity, we keep using claim sequences as inputs to the algorithms given below. This implies that (formally) the same public key may be given multiple times to the verification algorithm. In practice, it is obviously unnecessary to store the public key once per log entry, or give multiple copies of this key to the verification algorithms.

Definition 5.14 (Logging Scheme with List Verification)**.** A *logging scheme with list verification* $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$ is a tuple of five PPT algorithms, where:

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk}, \tau_0)$
: takes as input the security parameter κ and an a priori upper bound T (encoded in unary) on the number of epochs. It outputs a key pair $(\mathsf{sk}_1, \mathsf{pk})$, where sk_1 is the secret key for the first epoch, and an initial signature τ_0 for the empty log file $C_0 = ()$.

$\mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i) \to \tau_i$
: takes as input a secret key sk_t for epoch t, a claim sequence C_{i-1}, a corresponding signature τ_{i-1} and a message m_i. It outputs a signature τ_i for the new claim sequence $C_i := C_{i-1} \| (\mathsf{pk}, t, m_i)$.

$\mathsf{Update}(\mathsf{sk}_t, C, \tau) \to \mathsf{sk}_{t+1}$
: takes as input the secret key sk_t of period t, the current claim sequence C and a signature τ for C. If $t \geq T$ the output is undefined. If $t < T$ it computes the secret key sk_{t+1} for period $t+1$ and securely erases the old key sk_t. Update may modify C by appending additional claims to it, and τ may be modified arbitrarily.

$\mathsf{ValidEntries}(C, \tau) \to V$
: takes as input a claim sequence C of length $n \in \mathbb{N}_0$ and a signature τ for C and outputs a bit string $V \in \{0,1\}^n$ (also of length n). Claims $C[i]$ (for $i \in [n]$) with $V[i] = 1$ are taken to be valid, claims with $V[i] = 0$ are considered invalid.

$\mathsf{VerifyLog}(C, \tau) \to 0/1$
: outputs either 1, if C is deemed completely valid and authentic, or 0 otherwise.

We require that a logging scheme with list verification is correct as defined below.

The notion of *regular* signatures is defined analogously to the definition for categorized key-evolving audit log schemes (see Definition 4.9):

Definition 5.15 (Valid and Regular Signatures)**.** Let LS be a tuple as defined above, and $\kappa \in \mathbb{N}$. We say a signature τ is *valid* for a claim sequence C iff $\mathsf{VerifyLog}(C, \tau) = 1$. We say that τ is *valid for index i of C*, iff $\mathsf{ValidEntries}(C, \tau)[i] = 1$, where $i \in [\mathrm{len}(C)]$. In this case, we may also say that τ is *valid for the i-th claim in C*. When i and C are clear from the context, we may simply say that τ is *valid for the claim $c = C[i]$*.

Additionally, we say that τ is *regular* for C iff (C, τ) are in the image of the following process for some number of epochs $T = T(\kappa) \in \mathrm{poly}(\kappa)$, a log file length $l \in \mathbb{N}_0$, a log file $M = (m_1, \ldots, m_l)$, and a monotonically increasing sequence $(t_1, \ldots, t_{l+1})$ of epoch numbers (each in $[T]$):

1. Let $(\mathsf{sk}_1, \mathsf{pk}, \tau) \leftarrow \mathsf{KeyGen}(1^\kappa, 1^T)$, $t \coloneqq 1$, $C_0 \coloneqq ()$.
2. Iterate over all $i \in [l]$ in increasing order:
   a) While $t_i > t$, compute $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t, C_{i-1}, \tau)$ and set $t \coloneqq t + 1$. As above, Update may modify C_{i-1} and τ.
   b) Set $\tau \coloneqq \mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau, m_i)$.
   c) Set $c_i = (\mathsf{pk}, t_i, m_i)$ and $C_i \coloneqq C_{i-1} \,\|\, c_i$.
3. While $t_{l+1} > t$, keep setting $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t, C_l, \tau)$ and $t \coloneqq t + 1$.
4. Output (C_l, τ).

This concludes our definition of regular signatures. We now define *fault tolerance* for key-evolving logging schemes with list verification. The definition is analogous to the definition for key-evolving sequential aggregate signature schemes (see Section 5.3).

Definition 5.16 (Fault Tolerance)**.** Let LS be a tuple as defined above, C, C' be two claim sequences where C' contains at most d errors with respect to C, let τ be a regular signature for C, and $V \coloneqq \mathsf{ValidEntries}(C', \tau)$. LS is *tolerant against d errors* iff for all such claim sequences C, C', we have

$$V[i] = 1 \quad \text{for all } i \in [\min(\mathrm{len}(C), \mathrm{len}(C'))] \text{ where } C[i] = C'[i].$$

LS is *d-fault-tolerant* if it is tolerant against d errors. We say that LS is *fault-tolerant*, if it is d-fault-tolerant for some $d \geq 1$.

We ask that for a given claim sequence C and a signature τ, the behavior of $\mathsf{ValidEntries}$ and $\mathsf{VerifyLog}$ is consistent: If $\mathsf{VerifyLog}$ indicates that a claim sequence C is completely valid, then $\mathsf{ValidEntries}$ should identify all claims in C as valid.

Note that we do not require the converse property, since it is perfectly possible that all claims in a claim sequence are valid, but the entire claim sequence does not correspond to the "true" log file, i.e. if the log file has been truncated.

Definition 5.17 (Verification Consistency)**.** Let LS be a tuple as defined above. LS is *verification-consistent* iff for each claim sequence C and each signature τ with $\mathsf{VerifyLog}(C, \tau) = 1$, we have that $\mathsf{ValidEntries}(C, \tau) = 1^{\mathrm{len}(C)}$.

Finally, we may now define:

Definition 5.18 (Correctness and Robustness)**.** Let LS be a tuple as defined above. LS is *correct* iff it is verification-consistent and for all claim sequences C and all signatures τ regular for C, we have that τ is valid for C. If LS moreover is fault-tolerant, then it is *robust*.

5.4.1. Security Notion

As our next step, we define our security notion for logging schemes with list verification. The experiment of this notion is similar in spirit to the security experiment for logging schemes supporting excerpts, given in Definition 4.13 (see page 73). However, there is a notable difference:

We do not return log file signatures to the attacker by default. Observe that if the experiment returned the log file signature after each query to the attacker's Append oracle, then the attacker had obtained valid signatures for each intermediate state of the log file, and hence all truncation attacks had to be considered trivial. However modeling truncation attacks is one of the goals of our research. Thus we let the attacker (adaptively) specify the log file states for which to obtain a signature. More specifically, the attacker must explicitly request the log file signature for specific log file states via a separate $\mathsf{GetSignature}$ oracle. This may help an attacker in forging signatures, but truncation attacks to one of the states for which the attacker requested the signature are considered trivial after such a query.

Observe that in the security experiment for logging schemes supporting excerpts given in Definition 4.13 (see page 73) the extraction oracle implicitly realized the function of the $\mathsf{GetSignature}$ oracle.

At the end of the experiment, the attacker outputs a forgery. We require that the forged claim sequence is valid and non-trivial, *or* there is a valid and non-trivial claim.

Definition 5.19 (Forward-Secure Existential Unforgeability under Chosen Log Message Attacks)**.** For a log scheme with list verification $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$, a PPT adversary $\mathcal{A}$, the number of epochs T and the security parameter $\kappa \in \mathbb{N}$, the security experiment is defined as follows:

Setup Phase.

The experiment generates a key pair with an initial signature $(\mathsf{sk}_1, \mathsf{pk}, \tau_0) \coloneqq \mathsf{KeyGen}(1^\kappa, 1^T)$, and initializes the log file $C_0 \coloneqq ()$. It maintains an epoch counter t with an initial value of 1, initializes $i \coloneqq 1$, and then starts $\mathcal{A}$ with inputs pk, 1^T.

Query Phase.

During the query phase, $\mathcal{A}$ may adaptively issue queries to the following oracles:

LogAppend Oracle.

When the attacker specifies a message m_i, the experiment updates the signature via $\tau_i \coloneqq \mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i)$, where C_{i-1} is the current

log file and τ_{i-1} is the current signature. The experiment then appends the message m_i to the log file, i.e. it sets $C_i = C_{i-1} \| (\mathsf{pk}, t, m_i)$, and sets $i \coloneqq i + 1$. The experiment returns "ok" to $\mathcal{A}$.

Epoch Switching Oracle.
The NextEpoch oracle updates the secret key, the log and its signature via $\mathsf{sk}_{t+1} \coloneqq \mathsf{Update}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1})$, increments the epoch counter $t \coloneqq t + 1$ and returns "ok". It may be queried at most $T - 1$ times.

GetSignature Oracle.
Whenever $\mathcal{A}$ calls the GetSignature oracle, the experiment responds with the current signature τ_i of the log.

Break-In Phase.
Once $\mathcal{A}$ signals it is done with the query phase, the experiment enters the break-in phase. During the break-in phase, $\mathcal{A}$ is no longer allowed queries to the oracles provided during the query phase. Instead, the adversary may use a BreakIn oracle to obtain the current secret key sk_t. If $\mathcal{A}$ does, the experiment sets $t_{\mathsf{BreakIn}} \coloneqq t$. Otherwise, let $t_{\mathsf{BreakIn}} \coloneqq \infty$.

Forgery Phase.
$\mathcal{A}$ outputs a log file C^*, and a forged signature τ^* for C^*.

We say that C^* is *trivial*, iff:

- C^* equals the contents of some C_i during any of $\mathcal{A}$'s GetSignature oracle queries, or
- there is a claim $c^* = (\mathsf{pk}^*, t^*, m^*) \in C^*$ such that $\mathsf{pk}^* \neq \mathsf{pk}$, or
- $\mathcal{A}$ used the BreakIn oracle in some epoch t_{BreakIn} and C' is a prefix of C^*, where C' is the state of the log file after the epoch switch from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn} (including changes made by Update, if any). We let $C' = ()$ if $t_{\mathsf{BreakIn}} = 1$.

A claim $c^* = (\mathsf{pk}^*, t^*, m^*)$ is called *trivial* iff $\mathsf{pk}^* \neq \mathsf{pk}$, $t^* \geq t_{\mathsf{BreakIn}}$, $\mathcal{A}$ queried m^* at its LogAppend oracle during epoch t^*, or c^* is equal to a claim that was appended by one of the Update calls performed by the epoch switching oracle.

The experiment outputs 1 iff

- C^* is non-trivial and τ^* is valid for C^* or
- there is an index $i^* \in [\mathrm{len}(C^*)]$ such that τ^* is valid for index i^* of C^* and $c^* \coloneqq C^*[i^*]$ is non-trivial.

Otherwise, the experiment outputs 0.

We say that $\mathcal{A}$ *wins* the experiment, iff the experiment outputs 1, otherwise, $\mathcal{A}$ *loses* the experiment.

A logging scheme with list verification LS is said to be *forward-secure existentially unforgeable under chosen log message attacks* (*FS-EUF-CLMA-secure*) iff for all $T = T(\kappa) \in \text{poly}(\kappa)$ and all probabilistic polynomial time attackers $\mathcal{A}$:

$$\Pr[\mathcal{A} \text{ wins}] \leq \mathsf{negl}(\kappa).$$

Note that we consider forgeries where the attacker does not use the challenge public key out-of-scope for this security definition. As explained above, we only keep using the claim sequence notation for simplicity, but expect that in practice the public key is stored just once for all log entries. Thus, the definition given above formally requires the attacker to use the "correct" public key since forgeries where the attacker uses other keys are considered trivial.

5.4.2. Generic Construction

We now turn to our generic construction of a robust logging scheme. The scheme is built from an arbitrary fault-tolerant forward-secure SAS scheme and a plain forward-secure signature scheme. Much of the functionality of this scheme is inherited from the underlying fault-tolerant scheme, but our transformation adds message counters to protect against message reordering and epoch markers (together with signatures on the log file length) to achieve security against truncation attacks. Recall our assumption that messages are properly encoded before signing (see Section 2.4). This encoding must be injective, i.e. it must be guaranteed that no two distinct objects are mapped to the same bit string. This assumption is required for our proof of security.

We claim that the construction given below simultaneously achieves security, robustness and sub-linear storage overhead, if the applied encoding is injective, the underlying schemes are secure and correct, and the CFF given in Appendix A is used.

Let AS be a key-evolving SAS scheme with list verification and FS a key-evolving signature scheme. We define the logging scheme with list verification $\mathsf{LS} = (\mathsf{KeyGen}, \mathsf{Append}, \mathsf{Update}, \mathsf{ValidEntries}, \mathsf{VerifyLog})$ as follows:

$\mathsf{KeyGen}(1^\kappa, 1^T) \to (\mathsf{sk}_1, \mathsf{pk}, \tau_0)$
creates key pairs of the underlying schemes AS and FS as

$$(\mathsf{sk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{AS}}) \leftarrow \mathsf{AS}.\mathsf{KeyGen}(1^\kappa, 1^T),$$
$$(\mathsf{sk}_{\mathsf{FS}}, \mathsf{pk}_{\mathsf{FS}}) \leftarrow \mathsf{FS}.\mathsf{KeyGen}(1^\kappa, 1^T)$$

and returns $\mathsf{sk}_1 = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$ and $\mathsf{pk} = (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$. The signature for the empty log file $C_0 = ()$ is $\tau_0 := (\lambda, \mathsf{FS}.\mathsf{Sign}(\mathsf{sk}_{\mathsf{FS}}, 0))$.

$\mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i) \to \tau_i$
takes as input a secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$ for period t, a claim sequence $C_{i-1} = (c_1, \ldots, c_{i-1}) = ((\mathsf{pk}, t_j, m_j))_{j=1}^{i-1}$, its corresponding signature $\tau_{i-1} = (\sigma_{i-1}, s_{i-1})$

and a message m_i to sign. Let $C_{\mathsf{AS}} = ((\mathsf{pk}_{\mathsf{AS}}, t_j, (j, m_j)))_{j=1}^{i-1}$. Both signature components are obtained from the signature algorithms of AS and FS via

$$\sigma_i \coloneqq \mathsf{AS.AggSign}(\mathsf{sk}_{\mathsf{AS}}, C_{\mathsf{AS}}, \sigma_{i-1}, (i, m_i)) \text{ and}$$
$$s_i \coloneqq \mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}, i).$$

Then Append securely and irrecoverably erases the old signature s_{i-1} on the log file length. The resulting signature $\tau_i \coloneqq (\sigma_i, s_i)$ is returned.

$\mathsf{Update}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}) \to \mathsf{sk}_{t+1}$

takes as input the secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}, \mathsf{sk}_{\mathsf{FS}})$, a claim sequence C_{i-1} and a corresponding signature τ_{i-1}, and appends an epoch marker to the log file that is valid for the current epoch t, via

$$\tau_i \coloneqq \mathsf{Append}(\mathsf{sk}_t, C_{i-1}, \tau_{i-1}, m_i),$$

where $m_i \coloneqq (\text{“}\texttt{End of epoch:}\text{”}, t)$. It then updates the components of sk_t via $\mathsf{sk}'_{\mathsf{AS}} \coloneqq \mathsf{AS.Update}(\mathsf{sk}_{\mathsf{AS}})$ and $\mathsf{sk}'_{\mathsf{FS}} \coloneqq \mathsf{FS.Update}(\mathsf{sk}_{\mathsf{FS}})$. (These algorithms securely erase the old keys.) The new secret key is $\mathsf{sk}_{t+1} = (\mathsf{sk}'_{\mathsf{AS}}, \mathsf{sk}'_{\mathsf{FS}})$, the new claim sequence is $C_i = C_{i-1} \,\|\, (\mathsf{pk}, t, m_i)$, and the new signature is τ_i.

$\mathsf{ValidEntries}(C, \tau) \to V$

takes as input a claim sequence C and a signature $\tau = (\sigma, s)$ for C. It outputs $\mathsf{AS.Verify}(C_{\mathsf{AS}}, \sigma)$, where C_{AS} is the claim sequence generated from C by prepending the message number i to m_i and replacing the public key pk with the respective key $\mathsf{pk}_{\mathsf{AS}}$ for all claims in C. More precisely, if $C = ((\mathsf{pk}, t_i, m_i))_{i=1}^{n}$ (for some $n \in \mathbb{N}_0$), then $C_{\mathsf{AS}} = ((\mathsf{pk}_{\mathsf{AS}}, t_i, (i, m_i)))_{i=1}^{n}$.

$\mathsf{VerifyLog}(C, \tau) \to 0/1$

takes as input a claim sequence $C = (c_i)_{i=1}^{n} = ((\mathsf{pk}_i, t_i, m_i))_{i=1}^{n}$ (for some $n \in \mathbb{N}_0$) and a signature $\tau = (\sigma, s)$ for C. Firstly, it verifies that the public keys of all claims in C are equal. If there are different keys, then $\mathsf{VerifyLog}$ outputs 0.

Secondly, for all epoch markers c_i in C (that is, for all claims where m_i is not a bit string), $\mathsf{VerifyLog}$ checks if $m_i = (\text{“}\texttt{End of epoch:}\text{”}, t_i)$. Moreover, it verifies that for all $i \in [\mathrm{len}(C) - 1]$, we have

$$t_{i+1} = t_i + 1 \qquad \text{if } c_i \text{ is an epoch marker, and}$$
$$t_{i+1} = t_i \qquad \text{otherwise.}$$

If one of these conditions is not met, then $\mathsf{VerifyLog}$ returns 0.

Let t be the maximum of all t_i. Thirdly, the algorithm verifies the FS signature s using

$$b \coloneqq \mathsf{FS.Verify}((\mathsf{pk}_{\mathsf{FS}}, t, \mathrm{len}(C)), s).$$

If $b = 0$, it returns 0.

Finally, $\mathsf{VerifyLog}$ checks whether $\mathsf{ValidEntries}(C, \tau) = 1^{\mathrm{len}(C)}$, and returns 1 if so. Otherwise, it returns 0.

We defer the proof of correctness and robustness to Appendix B, since the proof is essentially straightforward, but a little lengthy. Here, we only claim:

Theorem 5.20 (Correctness and Robustness)**.** If FS is correct and AS is correct, then LS is correct. If moreover AS is fault-tolerant for a $d > 0$, then LS is d-fault-tolerant and hence robust.

We now prove the security of our scheme. Once again, recall our convention from Section 2.4 that all mathematical objects are encoded to bit strings in some uniquely invertible fashion before signing. Furthermore, observe that the VerifyLog algorithm verifies that the epoch markers t_i of the input claim sequence C form a monotonically increasing sequence, and there are no "gaps": The difference between two consecutive t_i is at most 1.

Theorem 5.21. Let LS be the logging scheme with list verification defined above and AS, FS be the underyling signature schemes. If there is a PPT adversary $\mathcal{A}$ who breaks the FS-EUF-CLMA-security of LS with success probability $\varepsilon_{\mathcal{A}}$, then there exists a PPT adversary $\mathcal{B}$ who breaks the FS-SAS-EUF-CMA-security of AS with success probability at least $\varepsilon_{\mathcal{B}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$, or there exists a PPT adversary $\mathcal{C}$ who breaks the FS-EUF-CMA-security of FS with success probability $\varepsilon_{\mathcal{C}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$.

Let us first give some overview and intuition about the proof. To win the security experiment, an attacker $\mathcal{A}$ must either truncate the log file to a state he has not seen the signature for, or create a valid signature for a log file modified w.r.t. an epoch before his break-in. If $\mathcal{A}$ truncates the log file without detection, he must create a new signature s for the length of the log file, which violates the security of FS. If $\mathcal{A}$ forges a signature for a log file modified w.r.t. a previous epoch, then $\mathcal{A}$ has broken the security of AS. Since we assume that both base schemes are secure, our resulting construction must be secure, too.

Proof. A FS-EUF-CLMA adversary $\mathcal{A}$ can adaptively query the three oracles Append, GetSignature and NextEpoch before he may break in, and then outputs a forgery (C^*, τ^*), where $\tau^* = (\sigma^*, s^*)$.

If $\mathcal{A}$ uses the BreakIn oracle during some epoch t_{BreakIn}, let C' be the state of the log file directly after $\mathcal{A}$'s last call to the NextEpoch oracle, switching from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn}. (Let $C' := ()$ if $\mathcal{A}$ did break in during epoch $t_{\mathsf{BreakIn}} = 1$.) We consider four different events:

- E_1 occurs, iff $\mathcal{A}$ wins the game, but $\mathsf{VerifyLog}(C^*, \tau^*) = 0$.
- E_2 occurs, iff $\mathcal{A}$ wins the game, $\mathsf{VerifyLog}(C^*, \tau^*) = 1$, and $\mathcal{A}$ sent GetSignature request while the length of the log file was $\mathrm{len}(C^*)$.
- E_3 occurs, iff $\mathcal{A}$ wins the game, $\mathsf{VerifyLog}(C^*, \tau^*) = 1$, $\mathcal{A}$ did not send a GetSignature request while the length of the log file was $\mathrm{len}(C^*)$, and the last claim in C^* has $t \geq t_{\mathsf{BreakIn}}$.

- E_4 occurs, iff $\mathcal{A}$ wins the game, $\mathsf{VerifyLog}(C^*, \tau^*) = 1$, $\mathcal{A}$ did not send a $\mathsf{GetSignature}$ request while the length of the log file was $\mathrm{len}(C^*)$, and the last claim in C^* has $t < t_{\mathsf{BreakIn}}$.

We have $\varepsilon_{\mathcal{A}} \leq \Pr[E_1]+\Pr[E_2]+\Pr[E_3]+\Pr[E_4]$ and thus $\Pr[E_1]+\Pr[E_2]+\Pr[E_3] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ or $\Pr[E_4] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$. In the following paragraphs $\mathsf{sk}^t_{\mathsf{AS}}$ and $\mathsf{sk}^t_{\mathsf{FS}}$ denote the secret keys of epoch t for the respective schemes.

Attack on the FS-SAS-EUF-CMA Security of AS.

First we construct a FS-SAS-EUF-CMA adversary $\mathcal{B}$ on AS, who uses a successful FS-EUF-CLMA adversary $\mathcal{A}$ and has to simulate the FS-EUF-CLMA security experiment for $\mathcal{A}$. The challenger in the FS-SAS-EUF-CMA-security experiment generates a key pair $(\mathsf{sk}^1_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{AS}}) \coloneqq \mathsf{AS.KeyGen}(1^\kappa, 1^T)$ and sends $\mathsf{pk}_{\mathsf{AS}}$ and the maximal number of epochs T (encoded in unary) to $\mathcal{B}$. $\mathcal{B}$ uses FS to generate a key pair $(\mathsf{sk}^1_{\mathsf{FS}}, \mathsf{pk}_{\mathsf{FS}}) \coloneqq \mathsf{FS.KeyGen}(1^\kappa, 1^T)$, and sets $\mathsf{pk} \coloneqq (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$. $\mathcal{B}$ initializes the log and signature it maintains towards $\mathcal{A}$ as $C_0 \coloneqq ()$, $\sigma_0 \coloneqq \lambda$ and sets $i \coloneqq 1$, $t \coloneqq 1$ and $\mathcal{L}_{\mathrm{FS}} \coloneqq (\mathsf{sk}^1_{\mathsf{FS}})$. $\mathcal{B}$ then and starts executing $\mathcal{A}$ with inputs pk and 1^T.

We describe how $\mathcal{B}$ simulates the three oracles and the break-in phase for $\mathcal{A}$:

LogAppend Oracle.

$\mathcal{A}$ sends a query m_i. $\mathcal{B}$ sets $C_i \coloneqq C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, (i, m_i))$ for the current period t. $\mathcal{B}$ then sends an $\mathsf{AggSign}$ query (i, m_i) with claim sequence C_{i-1} and signature σ_{i-1} to his challenger who responds with a signature σ_i. Finally, $\mathcal{B}$ sets $i \coloneqq i+1$ and sends the string "$\mathtt{ok}$" to $\mathcal{A}$.

Epoch Switching Oracle.

When $\mathcal{A}$ sends a $\mathsf{NextEpoch}$ query, $\mathcal{B}$ aborts if $t \geq T$, otherwise $\mathcal{B}$ sets $m_i \coloneqq$ ("$\mathtt{End\ of\ epoch:}$", t) and $C_i \coloneqq C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, (i, m_i))$. $\mathcal{B}$ obtains the signature σ_i for C_i from his $\mathsf{AS.AggSign}$ oracle the same way as before. $\mathcal{B}$ then sends an epoch switching query to the challenger, who computes $\mathsf{sk}^{t+1}_{\mathsf{AS}} \coloneqq \mathsf{AS.Update}(\mathsf{sk}^t_{\mathsf{AS}})$. $\mathcal{B}$ computes $\mathsf{sk}^{t+1}_{\mathsf{FS}} \coloneqq \mathsf{FS.Update}(\mathsf{sk}^t_{\mathsf{FS}})$ by its own. $\mathcal{B}$ sets $i \coloneqq i+1$, $t \coloneqq t+1$ and $\mathcal{L}_{\mathrm{FS}} \coloneqq \mathcal{L}_{\mathrm{FS}} \,\|\, \mathsf{sk}^{t+1}_{\mathsf{FS}}$ and returns "$\mathtt{ok}$".

GetSignature Oracle.

When $\mathcal{A}$ calls the $\mathsf{GetSignature}$ oracle, $\mathcal{B}$ determines the length i of the current claim sequence C_i and the period t_{last} of the last claim in C_i, which is either the current period t or $t-1$ (since an epoch switch always adds an end-of-epoch claim). $\mathcal{B}$ retrieves $\mathsf{sk}^{t_{\mathrm{last}}}_{\mathsf{FS}}$ from $\mathcal{L}_{\mathrm{FS}}$ and computes $s_i \coloneqq \mathsf{FS.Sign}(\mathsf{sk}^{t_{\mathrm{last}}}_{\mathsf{FS}}, i)$. $\mathcal{B}$ sends $\tau_i \coloneqq (\sigma_i, s_i)$ as the signature for C_i to $\mathcal{A}$.

Break-In Phase.

When $\mathcal{A}$ breaks in, $\mathcal{B}$ sets $t_{\mathsf{BreakIn}} \coloneqq t$ and sends the challenger a $\mathsf{BreakIn}$ request. $\mathcal{B}$ gets the current secret key $\mathsf{sk}^t_{\mathsf{AS}}$ and sends the current secret key $\mathsf{sk}_t = (\mathsf{sk}^t_{\mathsf{AS}}, \mathsf{sk}^t_{\mathsf{FS}})$ to $\mathcal{A}$.

If none of the events E_1, E_2, E_3 take place, then $\mathcal{B}$ aborts. Define $l^* \coloneqq \mathrm{len}(C^*)$, and let C^*_{AS} be the claim sequence which is obtained by prepending the message index i to m_i in each of the claims from C^* and replacing the public key pk by $\mathsf{pk}_{\mathsf{AS}}$. More formally, if $C^* = ((\mathsf{pk}, t_i^*, m_i^*))_{i=1}^{l^*}$, then let $C^*_{\mathsf{AS}} = ((\mathsf{pk}_{\mathsf{AS}}, t_i^*, (i, m_i^*)))_{i=1}^{l^*}$. $\mathcal{B}$ outputs $(C^*_{\mathsf{AS}}, \sigma^*)$. We now must show that $\mathcal{B}$'s output C^*_{AS} contains a non-trivial claim at some position i^* and σ^* is valid for the claim $C^*_{\mathsf{AS}}[i^*]$.

Valid and Non-Trivial Claim for E_1. If E_1 takes place, then τ^* is not valid for C^*, so $\mathcal{A}$ must have won the game by producing a signature that authenticates a non-trivial claim $c^* = C^*[i^*] = (\mathsf{pk}^*, t^*, m^*)$ (for $i^* \in [\mathrm{len}(C^*)]$). We will show that $c^*_{\mathsf{AS}} = C^*_{\mathsf{AS}}[i^*] = (\mathsf{pk}^*_{\mathsf{AS}}, t^*, (i^*, m^*))$ is a valid and non-trivial claim in the FS-SAS-EUF-CMA game.

To see that c^*_{AS} is valid, observe that $\mathsf{ValidEntries}(C^*, \tau^*)$ outputs the result of $\mathsf{AS.Verify}(C^*_{\mathsf{AS}}, \sigma^*)$. Hence, if τ^* is valid for position i^* of C^*, then σ^* must be valid for position i^* of C^*_{AS}, too, and thus c^*_{AS} is valid. We now verify that c^*_{AS} is non-trivial. Firstly, c^*_{AS} by construction uses $\mathsf{pk}_{\mathsf{AS}}$ as the public key. Secondly, since c^* is non-trivial we have that $t^* < t_{\mathsf{BreakIn}}$ in the FS-EUF-CLMA game. Since $\mathcal{B}$ queries its epoch switching oracle whenever $\mathcal{A}$ does and $\mathcal{B}$ breaks in iff $\mathcal{A}$ does, we also have $t^* < t_{\mathsf{BreakIn}}$ in the FS-SAS-EUF-CMA game. Finally, since c^* is non-trivial in the FS-EUF-CLMA game, $\mathcal{A}$ did not query m^* at the log appending oracle during epoch t^*, and c^* was not added during Update neither. Hence, $\mathcal{B}$ never queried (i^*, m^*) at its signature oracle. Thus, c^*_{AS} is in fact non-trivial.

Validity for E_2 and E_3. If E_2 or E_3 occur, then $\mathcal{A}$'s signature τ^* is valid for C^* (in the FS-EUF-CLMA experiment with LS). By the verification consistency of LS, this implies that $\mathsf{ValidEntries}(C^*, \tau^*) = 1^{\mathrm{len}(C^*)}$, and so σ^* is valid for *all* claims in C^*_{AS} (in the FS-SAS-EUF-CMA experiment with AS).

Non-Triviality for E_2 and E_3. We now need to show that C^*_{AS} contains a non-trivial claim. Let C_{last} be the state of the log file at the end of the FS-EUF-CLMA experiment simulated by $\mathcal{B}$. First note that all claims c in C^*_{AS} by construction have $\mathsf{pk}_{\mathsf{AS}}$ as public key. Hence, the first condition for the non-triviality is met by all of these claims. We now show that if either E_2 or E_3 happen, then there exists an index i^* such that $c^* \coloneqq C^*[i^*] \neq C_{\mathrm{last}}[i^*]$ and the epoch number t^* of c^* is less than t_{BreakIn}. This part of the proof depends on whether E_2 or E_3 occur. We begin with E_2.

Candidate Non-Trivial Claim for E_2. We show that there is an index $i^* \in [l^*]$ such that $C^*[i^*] \neq C_{\mathrm{last}}[i^*]$. Since E_2 occured, $\mathcal{A}$ did a $\mathsf{GetSignature}$ query while the log file maintained by the experiment had length l^*. Let C_{l^*} be the state of the log file during that query. Since $\mathcal{A}$'s forgery is non-trivial, C^* does not equal C_{l^*}. Thus, since C^* and C_{l^*} are of the same length but not equal, there must be an index i^* where C_{l^*} and C^* differ. Moreover, C_{l^*} is a prefix of C_{last}, so we have $C_{\mathrm{last}}[i^*] = C_{l^*}[i^*] \neq C^*[i^*]$. Hence,

we have shown that such an index i^* exists. In the following, let i^* denote the *least* index such that $C_{\text{last}}[i^*] = C^*[i^*]$.

Epoch Number for E_2. To see that $t^* < t_{\mathsf{BreakIn}}$, assume for the sake of a contradiction that $t^* \geq t_{\mathsf{BreakIn}}$. This immediately implies that $t_{\mathsf{BreakIn}} \neq \infty$, hence $\mathcal{A}$ broke in. Since i^* is the least index with $C_{\text{last}}[i^*] \neq C^*[i^*]$, we have that $C_{\text{last}}[i] = C^*[i]$ for all $i < i^*$. But if $t^* \geq t_{\mathsf{BreakIn}}$, then the epoch switch from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn} happened before index i^*, hence C^* is a mere continuation of C_{last} truncated to the most recent epoch switch, and hence C^* is trivial. This is a contradiction to the occurrence of the event E_2. Thus our assumption was false, and therefore $t < t_{\mathsf{BreakIn}}$. We have found an index i^* with the properties claimed above.

Candidate Non-Trivial Claim for E_3. We now argue that there is an index i^* with the claimed properties in the event of E_3 as well. If the epoch number of the last log entry in C^* is at least t_{BreakIn}, then obviously $t_{\mathsf{BreakIn}} \neq \infty$ and thus $\mathcal{A}$ broke in during the experiment. Thus, since C^* is non-trivial, C' is not a prefix of C^*, where C' is the state of the log file maintained by $\mathcal{B}$ directly after $\mathcal{A}$'s last $\mathsf{NextEpoch}$ query. Since τ^* is valid, the sequence of epoch numbers in C^* must be monotonically increasing. Let C'^* be the longest prefix of C^* such that all epoch numbers in C'^* have $t < t_{\mathsf{BreakIn}}$, and let C' be the state of the log file maintained by $\mathcal{B}$ after $\mathcal{A}$'s last $\mathsf{NextEpoch}$ query. Define $l'^* \coloneqq \text{len}(C'^*)$, $l' \coloneqq \text{len}(C')$.

We claim (again) there is an index $i^* \in [\min(l^*, l')]$ such that $C'[i^*] \neq C'^*[i^*] = C^*[i^*]$. Assume for the sake of a contradiction there is no such index. Then for all $i \in [\min(l^*, l')]$ we have $C'[i] = C^*[i]$. Hence, since C' is not a prefix of C^*, we have $l^* < l'$, and thus C^* is a prefix of C'. This implies, however, that the epoch number t of the last index of C^* must be at most the epoch number from the last log entry from C', which is $t_{\mathsf{BreakIn}} - 1$ (by construction). Hence, $t \leq t_{\mathsf{BreakIn}} - 1$, which contradicts $t \geq t_{\mathsf{BreakIn}}$, as required by E_3. Thus, we have shown there exists an index i^* with $C^*[i^*] \neq C'[i^*] = C_{\text{last}}[i^*]$. In the following, again let i^* be the *least* such index.

Epoch Number for E_3. We now show that the claim $c^* = C^*[i^*]$ has epoch number $t^* < t_{\mathsf{BreakIn}}$. Assume for the sake of a contradiction that $t^* \geq t_{\mathsf{BreakIn}}$. Since i^* is the least index such that $C^*[i^*]$ and $C_{\text{last}}[i^*]$ differ, we have that $C^*[i] = C_{\text{last}}$ for all $i < i^*$. Since $t^* \geq t_{\mathsf{BreakIn}}$, the epoch marker marking the transition from epoch $t_{\mathsf{BreakIn}} - 1$ to epoch t_{BreakIn} must have occured in C^* before index i^*, i.e. its index is $i^*_{\text{EM}} < i^* \leq \min(l^*, l')$. However, in C', this epoch marker occurs at index $i_{\text{EM}} = l'$, i.e. it is the last claim in C'. Since $i^*_{\text{EM}} < l' = i_{\text{EM}}$, C^* and C' must differ on position $i^*_{\text{EM}} < i^*$, which contradicts the fact that i^* is the first index where C^* and C' differ. Hence, our assumption was false, and $t^* < t_{\mathsf{BreakIn}}$.

Thus, regardless whether E_2 or E_3 take place, we have found an index i^* where $c^* = C^*[i^*] \neq C_{\text{last}}[i^*]$ and $t^* < t_{\mathsf{BreakIn}}$.

"Freshness." We now argue that $c^*_{\mathsf{AS}} := C^*_{\mathsf{AS}}[i^*] = (\mathsf{pk}^*_{\mathsf{AS}}, t^*, (i^*, m^*_{i^*}))$ is a non-trivial claim. We have already shown that all claims in C^*_{AS} have the public key $\mathsf{pk}^*_{\mathsf{AS}} = \mathsf{pk}_{\mathsf{AS}}$ (by construction), hence the first condition for a non-trivial claim is met. Moreover, we have shown that the claim $c^* = C^*[i^*]$ has $t^* < t_{\mathsf{BreakIn}}$. Since c^*_{AS} has the same epoch number t^*, this applies to c^*_{AS}, too. Thus, in order to show that c^*_{AS} is in fact non-trivial, it only remains to show that $\mathcal{B}$ did not query $(i^*, m^*_{i^*})$ at its signing oracle during epoch t^*.

Note that for each i $\mathcal{B}$ only performs a single request to its signing oracle, namely for (i, m_i). Moreover, the claims in C_{AS} directly correspond to $\mathcal{B}$'s queries to the signature oracle for AS: For each claim $c = (\mathsf{pk}_{\mathsf{AS}}, t, (i, m_i))$ in C_{AS}, $\mathcal{B}$ performed the query (i, m_i) during epoch t. Hence, only the i^*-th query of $\mathcal{B}$ can possibly equal $(i^*, m^*_{i^*})$. However, we have shown that $C^*[i^*] \neq C_{\text{last}}[i^*]$. Since C^* is non-trivial, the public key of C^* is the challenge public key pk, and thus these two claims must differ in their epoch number or their message. Hence, $C_{\mathsf{AS}}[i^*]$ and $C^*_{\mathsf{AS}}[i^*]$ must differ in their epoch number or their message, too. We have thus shown that $\mathcal{B}$ did not query the signature oracle for $(i^*, m^*_{i^*})$ during epoch t^*, which concludes the proof that $C^*_{\mathsf{AS}}[i^*]$ is non-trivial.

Summary of $\mathcal{B}$. We have shown that if E_1, E_2 or E_3 occur, then $\mathcal{B}$'s output contains a valid and non-trivial claim, hence $\mathcal{B}$ wins the FS-SAS-EUF-CMA experiment. Since $\mathcal{B}$ perfectly simulates the FS-EUF-CLMA experiment for $\mathcal{A}$, we thus have

$$\varepsilon_{\mathcal{B}} := \Pr[\mathcal{B} \text{ wins}] \geq \Pr[E_1] + \Pr[E_2] + \Pr[E_3].$$

Moreover, $\mathcal{B}$ only requires polynomial time (in κ), if $\mathcal{A}$ is a PPT algorithm.

Attack on the FS-EUF-CMA-security of FS.

Next, we construct a FS-EUF-CMA adversary $\mathcal{C}$ on FS, who uses a successful FS-EUF-CLMA attacker $\mathcal{A}$ and has to simulate the FS-EUF-CLMA security experiment for $\mathcal{A}$. The challenger in the FS-EUF-CMA security experiment generates a key pair $(\mathsf{sk}^1_{\mathsf{FS}}, \mathsf{pk}_{\mathsf{FS}}) \leftarrow \mathsf{FS.KeyGen}(1^\kappa, 1^T)$ and sends $\mathsf{pk}_{\mathsf{FS}}$ and the maximal number of epochs T to $\mathcal{C}$. $\mathcal{C}$ uses the AS scheme, generates a key pair $(\mathsf{sk}^1_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{AS}}) \leftarrow \mathsf{AS.KeyGen}(1^\kappa, 1^T)$ and forwards $\mathsf{pk} := (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$ and 1^T to $\mathcal{A}$. $\mathcal{C}$ initializes the log and signature it maintains towards $\mathcal{A}$ as $C_0 := ()$, $\sigma_0 := \lambda$ and sets $i := 1$, $t := 1$, $t' := 1$, $\mathcal{L}_{\text{FS}} := \emptyset$. $\mathcal{C}$ then starts executing $\mathcal{A}$. We describe how $\mathcal{C}$ simulates the three oracles and the break-in phase for $\mathcal{A}$:

LogAppend Oracle.

When $\mathcal{A}$ sends a log appending query m_i, $\mathcal{C}$ sets $C_i := C_{i-1} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t, (i, m_i))$ for the current period t and $\sigma_i := \mathsf{AS.AggSign}(\mathsf{sk}^t_{\mathsf{AS}}, C_{i-1}, \sigma_{i-1}, (i, m_i))$. Then $\mathcal{C}$ sets $i := i + 1$ and returns the string "$\mathtt{ok}$" to $\mathcal{A}$.

Epoch Switching Oracle.

When $\mathcal{A}$ sends a $\mathsf{NextEpoch}$ query, $\mathcal{C}$ stops if $t \geq T$ and outputs $\perp$, otherwise $\mathcal{C}$ sets

$m_i \coloneqq$ ("`End of epoch:`", t) and $C_i \coloneqq C_{i-1} \| (\mathsf{pk}_{\mathsf{AS}}, t, (i, m_i))$, and computes σ_i in the same way as before. $\mathcal{C}$ computes $\mathsf{sk}_{\mathsf{AS}}^{t+1} \coloneqq \mathsf{AS.Update}(\mathsf{sk}_{\mathsf{AS}}^{t})$ by its own, sets $i \coloneqq i+1, t \coloneqq t+1$ and returns "ok".

GetSignature Oracle.
When $\mathcal{A}$ calls the GetSignature oracle, $\mathcal{C}$ determines the length i of the current claim sequence C_i. If $\mathcal{L}_{\mathrm{FS}}$ contains a tuple (i, s) for some signature s, then $\mathcal{C}$ returns (σ_i, s). Otherwise, $\mathcal{C}$ determines the period t_{last} of the last claim in C_i. If $t_{\mathrm{last}} - t' =: d \neq 0$, then $\mathcal{C}$ sends d NextEpoch queries to his challenger, who computes $\mathsf{sk}_{\mathsf{FS}}^{t_{\mathrm{last}}}$ by updating the current key $\mathsf{sk}_{\mathsf{FS}}^{t'}$ d times. $\mathcal{C}$ sets $t' \coloneqq t_{\mathrm{last}}$. Then $\mathcal{C}$ sends a signature query $m = i$ to his challenger, who responds with $s_i \coloneqq \mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}^{t_{\mathrm{last}}}, i)$. $\mathcal{C}$ stores (i, s_i) in $\mathcal{L}_{\mathrm{FS}}$ by setting $\mathcal{L}_{\mathrm{FS}} \coloneqq \mathcal{L}_{\mathrm{FS}} \cup \{(i, s_i)\}$. Finally, $\mathcal{C}$ returns $\tau_i \coloneqq (\sigma_i, s_i)$ as the signature for C_i to $\mathcal{A}$.

Break-In Phase.
When $\mathcal{A}$ sends a BreakIn request, $\mathcal{C}$ sets $t_{\mathsf{BreakIn}} \coloneqq t$. If $t - t' =: d \neq 0$, then $\mathcal{C}$ sends d NextEpoch queries to his challenger. After that, $\mathcal{C}$ sends a BreakIn request. The FS-EUF-CMA experiment returns the current secret key $\mathsf{sk}_{\mathsf{FS}}^{t}$ to $\mathcal{C}$, who sends the current secret key $\mathsf{sk}_t = (\mathsf{sk}_{\mathsf{AS}}^{t}, \mathsf{sk}_{\mathsf{FS}}^{t})$ to $\mathcal{A}$.

If event E_4 does not occur, then $\mathcal{C}$ aborts. If event E_4 does take place, then $\mathcal{A}$ outputs a valid and non-trivial forgery (C^*, τ^*), where the length $l^* \coloneqq \mathrm{len}(C^*)$ of C^* was never the current length of the log file during any GetSignature query by $\mathcal{A}$ and the epoch number t^* of the last claim in C^* is less than t_{BreakIn}. Let $\tau^* = (\sigma^*, s^*)$. $\mathcal{C}$ outputs (l^*, t^*, s^*) as its forgery in the FS-EUF-CMA experiment. We need to show that this forgery is valid and non-trivial.

Validity. Since $\mathcal{A}$'s forgery is valid, we must have that $\mathsf{FS.Verify}((\mathsf{pk}_{\mathsf{FS}}, t^*, l^*), s^*) = 1$, by the construction of the VerifyLog algorithm. Hence, s^* is valid for $(\mathsf{pk}_{\mathsf{FS}}, t^*, l^*)$.

Non-Triviality. We now argue that $\mathcal{C}$'s output is non-trivial (in the FS-EUF-CMA game). Firstly, $\mathcal{C}$ only performed signature queries for those lengths l for which $\mathcal{A}$ did a signature query. Hence, if event E_4 occurs, then $\mathcal{C}$ never performed a signature query for the length l^*. Secondly, since E_4 occured, we have $t^* < t_{\mathsf{BreakIn}}$, as required.

Summary of $\mathcal{C}$. We have shown that if E_4 takes place, then $\mathcal{C}$ wins the FS-EUF-CMA experiment. Since $\mathcal{C}$ perfectly simulates the FS-EUF-CLMA experiment for $\mathcal{A}$, we thus have

$$\varepsilon_{\mathcal{C}} \coloneqq \Pr[\mathcal{C} \text{ wins}] \geq \Pr[E_4].$$

Moreover, $\mathcal{C}$ runs in polynomial time (in κ) if $\mathcal{A}$ does.

Summary of the Proof. In total, we have shown $\varepsilon_{\mathcal{B}} \geq \Pr[E_1] + \Pr[E_2] + \Pr[E_3]$ and $\varepsilon_{\mathcal{C}} \geq \Pr[E_4]$. Hence, since we must have $\Pr[E_1] + \Pr[E_2] + \Pr[E_3] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ or $\Pr[E_4] \geq \frac{\varepsilon_{\mathcal{A}}}{2}$,

we have $\varepsilon_{\mathcal{B}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$ or $\varepsilon_{\mathcal{C}} \geq \frac{\varepsilon_{\mathcal{A}}}{2}$, as claimed. Moreover, $\mathcal{B}$ and $\mathcal{C}$ run in time polynomial in κ. □

Corollary 5.22. If AS is FS-SAS-EUF-CMA-secure and FS is FS-EUF-CMA-secure then LS is FS-EUF-CLMA-secure.

5.5. Implementation and Performance Results

We implemented our generic construction from Section 5.4.2 and conducted various benchmarks. Our scheme uses the BGLS-FS-SAS scheme [MT07a] and the BM-FSS scheme [BM99]. Our results are shown in Table 5.1.

Our implementation is written in C++11, and is available from [Har19a]. For the BM-FSS scheme, we chose a modulus size of 1024 bits, roughly equivalent to a security level of 80 bit. The BGLS scheme was instantiated using elliptic curve groups with 160 bits group size, and the base field had 1024 bits. We used an instantiation of the cover-free family based on polynomials, described in Appendix A. For a CFF supporting $n = 100$, 1000, and 10000 messages, we chose the field size $q = 5$, 11, and 23, respectively, and fixed the polynomial degree at $k = 2$. This led to $d = 2, 5$ and 11, respectively. (The resulting CFFs were slightly larger than required: They supported 125, 1331, and 12167 messages, respectively.) Whenever a hash function was needed, we used SHA-256. We used a constant string of 200 bytes for all messages.

Our experiments were conducted on a laptop computer with an Intel Core i5-2430M CPU [Int] with a clock rate of 2.4 GHz. (Our implementation is not parallelized and therefore did not make use of the additional processor cores.) The processor has private (per-core) caches of 128 KB (Level 1) and 512 KB (Level 2), and a shared Level 3 Cache of 3072 KB [Int12, Section 1.1] The system was equipped with 5.7 GiB of RAM and running a 64-bit desktop version of the Fedora 23 GNU/Linux operating system, equipped with Linux Kernel version 4.4.9-300. All code was compiled with the GNU C Compiler (version 5.3.1) and optimization level set to `-O2`. We used Shoups NTL library [Sho] (version 9.4.0) for the implementation of the BM-FSS scheme and the PBC library [Lyn] (version 0.5.14) for the implementation of the BGLS-FS-SAS scheme.

Methodology.

For our experiments, we defined several sets of processes. Each process was repeated three times. The averages and standard deviations shown in Table 5.1 have therefore been computed from a sample of size 3.

For the first set of processes, we called the KeyGen algorithm with the given parameter T and measured its total run-time. In the second set, we created a random key for T epochs, and then measured the run-time of updating the key T times, without computing any signatures. Table 5.1 shows the average run-time per invocation of Update.

The third process consisted of creating a key-pair valid for n epochs, and then calling the AggSign algorithm n times, switching epochs every ℓ messages. For each epoch

Table 5.1.: Runtimes of our robust secure logging schemes based on the BGLS-FS-SAS from [Ma08]. See the methodology section for an explanation of this table.

Algorithm	**Parameter**		ℓ	**avg [ms]**	**STD [ms]**
KeyGen	$T =$	100		451	25.7
		1000		3822	38.5
		10 000		38 053	241
Update	$T =$	10 000		18.6	0.033
AggSign + Update	$n =$	100	10	67.2	0.80
		100	100	60.5	1.63
		1000	10	67.5	0.014
		1000	100	60.4	0.048
		1000	1000	59.7	0.019
Verify	$n =$	100	10	129	1.50
		100	100	23.8	0.49
		1000	10	271	2.05
		1000	100	227	1.87
		1000	1000	22.5	0.15

switch, we created and signed an epoch marker first, and then updated the secret key. The process also included signing the current counter value with a forward-secure digital signature scheme and updating that scheme. The time shown in Table 5.1 is the total time of all signing and updating operations, divided by the number of messages, so it represents the average time needed for adding a single log entry to the log file. The standard deviation was computed over the average signing time in each run.

The measurements in the last set of processes were obtained by calling Verify after a completion of a process from the third set. The time given in Table 5.1 is an average of the run-time of three executions divided by the number of messages that were verified. Hence, it represents the average verification time per message. The standard deviation was computed over the run-times of an individual execution divided by n. We did not consider invalid signatures in our experiments.

5.6. Conclusion

Cryptographic logging schemes must be able to reliably detect modifications to log files. However, they also should be "tolerant" to modifications, in the sense that even if some log entries have been illicitly manipulated, the logging system should still be able to provide assurance of authenticity and integrity of log records which have not been modified. We have introduced the notion of *robustness* for logging schemes, which captures this requirement, and for the first time presented logging scheme which simultaneously is robust and has sub-linear storage overhead.

The scheme utilizes the approach outlined in Section 4.3 to achieve security against truncation attacks, and is proven to be secure according to our notion.

Finally, we evaluated the performance of a prototype implementation of our space-efficient and truncation-resistant robust secure logging scheme.

6. Summary

This thesis has introduced several new techniques for the construction of cryptographic logging schemes which can be used to detect retroactive modifications of log files, as well as cryptanalyzed three schemes proposed in the literature. The contributions offered by this thesis are briefly summarized below.

Logging with Excerpts. This thesis introduced a logging scheme which can create excerpts from log files, which can not only be checked for integrity and authenticity, but also for completeness. To the best of the author's knowledge, this scheme was the first scheme in the standalone model where completeness of excerpts can be checked by the verifier. The notion of completeness is formalized in the security notion given in Section 4.4.3, and the SALVE scheme given in Section 4.5 provably satisfies this notion.

Combining Aggregation with Robustness. Additionally, this thesis presented a new technique for constructing logging schemes in the standalone model which have sub-linear storage overhead *and* can successfully verify log entries even if other log entries have been modified. This thesis constructed a logging scheme using this technique, and showed that this scheme is secure and robust.

Truncation Security. This thesis moreover presented new formal security notions capturing truncation security and introduced a new technique to obtain logging schemes which can detect truncations of log files. This technique was applied to the previously described logging schemes, and the resulting schemes were proven secure according to our notions.

Attacks on Logging Schemes. Furthermore, we showed a total of four attacks on LogFAS [YPR12b], and the BM- and AR-FssAgg schemes [Ma08]. The attacks are entirely practical, and completely break the respective schemes. We have analyzed the security proof of the schemes and pointed out errors in them, resolving the contradiction between our attacks and the supposedly proven security of the schemes.

6.1. Open Questions and Future Work

We briefly point out some open problems and suggestions for future work.

Full Truncation Security. The schemes introduced in this thesis as well as certain schemes from the literature [MT08; MT09] only offer security against the deletion of log entries added before the break-in epoch. To the best of the author's knowledge, there is no logging scheme in the standalone model which offers *full* truncation security, i.e., which can protect against deletion of log entries added during the break-in epoch.

While it is possible to side-step this limitation of existing schemes by simply performing an epoch switch each time after a log entry has been recorded, the resulting schemes must typically support a very large number of epochs, incurring higher costs in computational resources such as runtime and storage.

In particular, the construction of an aggregate signature scheme which can offer protection against removal of individual signatures from aggregate signatures might lead to a scheme with full truncation security. In the context of secure logging, one would additionally require that the removal of individual signatures is intractable even given the current secret key.

Better Confidentiality for Logging with Excerpts. As discussed in Remark 4.19, the SALVE scheme introduced in Section 4.5 does not conceal how many log entries were added to which category during which epoch. Enhancing our scheme with some form of encryption (or designing a new scheme which can support excerpts and offer stricter confidentiality) is an interesting open problem.

Robustness to a Broader Range of Changes. Section 5.3.4 already discussed some limitations of the notion of "changes" tolerated by our fault-tolerant sequential aggregate signature scheme given in Section 5.3 and hence by our robust logging scheme in Section 5.4.

Section 5.3.4 proposed an approach that might be followed towards generalizing our scheme such that it offers more resistance to such changes. Analyzing this approach is left as future work.

Designing a fault-tolerant (sequential) aggregate signature scheme following a different approach than the one taken in this thesis might lead to other solutions to this problem, but is outside of the scope of this thesis.

New Forward-Secure Sequentially Aggregate Signature Schemes. Considering that the BM- and AR-FssAgg signature schemes are broken by the attacks presented in Section 3.3, the author is not aware of any secure constructions of forward-secure sequential aggregate signature schemes except for the BGLS-FssAgg scheme outlined in Section 2.3.2. Designing new forward-secure sequentially aggregate signature schemes is an important topic for future research.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Jörn Müller-Quade, for giving me the opportunity to do a doctorate under his supervision. He has granted me much more freedom than I might ever have asked for, yet was always ready to support me in what I did. This thesis would not have been possible without his trust and support.

I'd also like to thank Professor Tibor Jager, whose lecture on digital signatures has shaped my way of thinking about signatures and sparked my interest for authentication techniques. I'm furthermore indebted to him for taking on the effort of reviewing and assessing this thesis, as well as attending and grading my oral examination.

I'm grateful to my co-authors Brandon Broadnax, Nico Döttling, Dominik Hartmann, Max Hoffmann, Björn Kaidel, Alexander Koch, Jessica Koch, Matthias Nagel, Andy Rupp and Professor Jörn Müller-Quade for interesting discussions, the relaxed but productive research atmosphere and good cooperation.

My gratitude furthermore extends to Julia Hesse for bringing up the initial research question which led my coauthors and me to develop the techniques presented in [Har+16], and ultimately led to the results given in Chapter 5. I'd also like to thank Alexander Koch for questioning the security proof of the BM-FssAgg scheme, which was the starting point for my research presented in Section 3.3.

I'm also grateful to my parents, my brother, Sven Maier, Michael Klooß, and Björn Kaidel who provided useful feedback during the preparation of this thesis. I would furthermore like to thank Alexander Koch, whose valuable feedback and unrelenting attention to detail has helped me a lot during the preparation of my research papers as well as this thesis.

Finally, I'd like to thank all my current and former colleagues who made the daily work at KIT so pleasant. They will my reason to treasure my memories of working at KIT.

Bibliography

[Acc09] R. Accorsi. "Safe-Keeping Digital Evidence with Secure Logging Protocols: State of the Art and Challenges". In: *Fifth International Conference on IT Security Incident Management and IT Forensics*. 2009, pp. 94–110. ISBN: 978-0-7695-3807-5. DOI: 10.1109/IMF.2009.18.

[ADR02] J. H. An, Y. Dodis, and T. Rabin. "On the Security of Joint Signature and Encryption". In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by L. R. Knudsen. Lecture Notes in Computer Science 2332. Springer Berlin Heidelberg, 2002, pp. 83–107. ISBN: 978-3-540-46035-0. DOI: 10.1007/3-540-46035-7_6.

[AMN01] M. Abdalla, S. Miner, and C. Namprempre. "Forward-Secure Threshold Signature Schemes". In: *Topics in Cryptology — CT-RSA 2001*. Ed. by D. Naccache. Lecture Notes in Computer Science 2020. Springer Berlin Heidelberg, 2001, pp. 441–456. ISBN: 978-3-540-41898-6. DOI: 10.1007/3-540-45353-9_32.

[AP13] N. AlFardan and K. G. Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. 2013. URL: http://www.isg.rhul.ac.uk/tls/lucky13.html (visited on 08/13/2019).

[AR00] M. Abdalla and L. Reyzin. "A New Forward-Secure Digital Signature Scheme". In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by T. Okamoto. Lecture Notes in Computer Science 1976. Springer Berlin Heidelberg, 2000, pp. 116–129. ISBN: 978-3-540-41404-9. DOI: 10.1007/3-540-44448-3_10.

[AW92] W. A. Adkins and S. H. Weintraub. *Algebra: An Approach via Module Theory*. 1st ed. Graduate Texts in Mathematics 136. Springer, 1992. ISBN: 0-387-97839-9; 3-540-97839-9. See also [Wei].

[BM99] M. Bellare and S. K. Miner. "A Forward-Secure Digital Signature Scheme". In: *Advances in Cryptology — CRYPTO' 99*. Ed. by M. Wiener. Lecture Notes in Computer Science 1666. Springer Berlin Heidelberg, 1999, pp. 431–448. ISBN: 978-3-540-66347-8. DOI: 10.1007/3-540-48405-1_28.

[Bon+03] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. "Aggregate and Verifiably Encrypted Signatures from Bilinear Maps". In: *Advances in Cryptology — EUROCRYPT 2003*. Ed. by E. Biham. Lecture Notes in Computer Science 2656. Springer Berlin Heidelberg, 2003, pp. 416–432. ISBN: 978-3-540-14039-9. DOI: 10.1007/3-540-39200-9_26.

[Bow+14] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. "PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging". In: *Research in Attacks, Intrusions and Defenses, RAID 2014*. Ed. by A. Stavrou, H. Bos, and G. Portokalidis. Lecture Notes in Computer Science 8688. Springer, 2014, pp. 46–67. ISBN: 978-3-319-11378-4. DOI: 10.1007/978-3-319-11379-1_3.

[Boy+06] X. Boyen, H. Shacham, E. Shen, and B. Waters. "Forward-secure Signatures with Untrusted Update". In: *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2006, pp. 191–200. ISBN: 1-59593-518-5. DOI: 10.1145/1180405.1180430.

[BR93] M. Bellare and P. Rogaway. "Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols". In: *CCS '93: Proceedings of the 1st ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 1993, pp. 62–73. ISBN: 0-89791-629-8. DOI: 10.1145/168588.168596.

[Bro+16] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. *Concurrently Composable Security With Shielded Super-polynomial Simulators*. Cryptology ePrint Archive, Report 2016/1043. https://eprint.iacr.org/2016/1043. 2016.

[Bro+17] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. "Concurrently Composable Security with Shielded Super-Polynomial Simulators". In: *Advances in Cryptology – EUROCRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer International Publishing, 2017, pp. 351–381. ISBN: 978-3-319-56620-7. DOI: 10.1007/978-3-319-56620-7_13.

[Bul+14] A. Buldas, A. Truu, R. Laanoja, and R. Gerhards. "Efficient Record-Level Keyless Signatures for Audit Logs". In: *Secure IT Systems*. Ed. by K. Bernsmed and S. Fischer-Hübner. Springer International Publishing, 2014, pp. 149–164. ISBN: 978-3-319-11599-3. DOI: 10.1007/978-3-319-11599-3_9.

[Bun19] Bundesamt für Sicherheit in der Informationstechnik. *IT-Grundschutz-Kompendium*. 2nd ed. Bundesanzeiger Verlag, 2019. ISBN: 978-3-8462-0906-6. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2019.html (visited on 08/13/2019).

[BY03] M. Bellare and B. Yee. "Forward-Security in Private-Key Cryptography". In: *Topics in Cryptology — CT-RSA 2003*. Ed. by M. Joye. Lecture Notes in Computer Science 2612. Springer Berlin Heidelberg, 2003, pp. 1–18. ISBN: 978-3-540-00847-7. DOI: 10.1007/3-540-36563-X_1.

[BY97] M. Bellare and B. S. Yee. *Forward Integrity for Secure Audit Logs*. Tech. rep. University of California at San Diego, 1997.

[CC12] Common Criteria Editorial Board, ed. *Common Criteria for Information Technology Security Evaluation, Part 2*. Version 3.1 R4. 2012. URL: https://www.commoncriteriaportal.org/cc/ (visited on 08/13/2019).

[CN03] J. Coron and D. Naccache. "Boneh et al.'s k-Element Aggregate Extraction Assumption Is Equivalent to the Diffie-Hellman Assumption". In: *Advances in Cryptology — ASIACRYPT 2003*. Ed. by C. Laih. Lecture Notes in Computer Science 2894. Springer, 2003, pp. 392–397. ISBN: 3-540-20592-6. DOI: 10.1007/978-3-540-40061-5_25.

[CW09] S. A. Crosby and D. S. Wallach. "Efficient Data Structures for Tamper-Evident Logging". In: *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 2009, pp. 317–334. ISBN: 978-1-931971-69-0. URL: http://dl.acm.org/citation.cfm?id=1855768.1855788.

[DH76] W. Diffie and M. Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[FK95] A. Futoransky and E. Kargieman. "VCR y PEO: Dos Protocolos Criptográficos Simples". In: *25 Jornadas Argentinas de Informática e Investigación Operativa*. 1995. URL: https://www.coresecurity.com/sites/default/private-files/publications/2016/05/2Protocolos.pdf (visited on 08/13/2019).

[FK98] A. Futoransky and E. Kargieman. *VCR and PEO Revised*. 1998. URL: https://www.coresecurity.com/corelabs-research/publications/peo-revised (visited on 08/13/2019).

[FLS12] M. Fischlin, A. Lehmann, and D. Schröder. "History-Free Sequential Aggregate Signatures". In: *Security and Cryptography for Networks*. Ed. by I. Visconti and R. D. Prisco. Lecture Notes in Computer Science 7485. Springer, 2012, pp. 113–130. ISBN: 978-3-642-32927-2. DOI: 10.1007/978-3-642-32928-9_7.

[GMR84] S. Goldwasser, S. Micali, and R. L. Rivest. "A 'Paradoxical' Solution To The Signature Problem". In: *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1984, pp. 441–448. ISBN: 0-8186-0591-X. DOI: 10.1109/SFCS.1984.715946.

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-message Attacks". In: *SIAM Journal on Computing* 17.2 (1988), pp. 281–308. ISSN: 0097-5397. DOI: 10.1137/0217017.

[GR95] B. Guttman and E. A. Roback. *An Introduction to Computer Security: The NIST Handbook*. NIST Special Publication 800-12. US National Institute of Science and Technology, 1995. DOI: 10.6028/NIST.SP.800-12.

[GS02] C. Gentry and A. Silverberg. "Hierarchical ID-Based Cryptography". In: *Advances in Cryptology — ASIACRYPT 2002*. Ed. by Y. Zheng. Lecture Notes in Computer Science 2501. Springer Berlin Heidelberg, 2002, pp. 548–566. ISBN: 978-3-540-36178-7. DOI: 10.1007/3-540-36178-2_34.

[Har+16] G. Hartung, B. Kaidel, A. Koch, J. Koch, and A. Rupp. "Fault-Tolerant Aggregate Signatures". In: *Public-Key Cryptography — PKC 2016*. Ed. by C. Cheng, K. Chung, G. Persiano, and B. Yang. Lecture Notes in Computer Science 9614. Springer, 2016, pp. 331–356. ISBN: 978-3-662-49383-0. DOI: 10.1007/978-3-662-49384-7_13.

[Har+17a] G. Hartung, M. Hoffmann, M. Nagel, and A. Rupp. "BBA+: Improving the Security and Applicability of Privacy-Preserving Point Collection". In: *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017, pp. 1925–1942. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134071.

[Har+17b] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann. "Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures". In: *Provable Security*. Ed. by T. Okamoto, Y. Yu, M. H. Au, and Y. Li. Lecture Notes in Computer Science 10592. Springer International Publishing, 2017, pp. 87–106. ISBN: 978-3-319-68637-0. DOI: 10.1007/978-3-319-68637-0_6.

[Har13] G. Hartung. "Cryptographic primitives from the LPN assumption". MA thesis. Karlsruhe Institute of Technology, 2013.

[Har16a] G. Hartung. "Secure Audit Logs with Verifiable Excerpts". In: *Topics in Cryptology — CT-RSA 2016*. Ed. by K. Sako. Lecture Notes in Computer Science 9610. Springer, 2016, pp. 183–199. ISBN: 978-3-319-29484-1. DOI: 10.1007/978-3-319-29485-8_11.

[Har16b] G. Hartung. *Secure Audit Logs with Verifiable Excerpts – Full Version*. Cryptology ePrint Archive, Report 2016/283. https://eprint.iacr.org/2016/283. 2016.

[Har17] G. Hartung. "Attacks on Secure Logging Schemes". In: *Financial Cryptography and Data Security*. Ed. by A. Kiayias. Springer International Publishing, 2017, pp. 268–284. ISBN: 978-3-319-70972-7. DOI: 10.1007/978-3-319-70972-7_14.

[Har19a] D. Hartmann. *Implementation of Robust/Fault-Tolerant Secure Logging Scheme described in "Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures"*. See [Har+17b]. 2019. DOI: 10.5445/IR/1000100165.

[Har19b] G. Hartung. *Implementation of the Attacks on the BM-FssAgg and AR-FssAgg Signature Schemes*. See [Har17]. 2019. DOI: 10.5445/IR/1000096061.

[Hol06] J. E. Holt. "Logcrypt: Forward Security and Public Verification for Secure Audit Logs". In: *Proceedings of the 2006 Australasian Workshops on Grid Computing and e-Research – Volume 54*. ACSW Frontiers '06. Australian Computer Society, Inc., 2006, pp. 203–211. ISBN: 1-920-68236-8.

[HR17] R. Haynes and J. S. Roberts. *Automated Trading in Futures Markets — Update*. 2017. URL: https://www.cftc.gov/About/EconomicAnalysis/ResearchPapers/ssLINK/oce_automatedtrading_update (visited on 01/27/2020).

[HWI03] F. Hu, C.-H. Wu, and J. D. Irwin. *A New Forward Secure Signature Scheme using Bilinear Maps*. Cryptology ePrint Archive, Report 2003/188. 2003. URL: https://eprint.iacr.org/2003/188.

[Ida+15] T. B. Idalino, L. Moura, R. F. Custódio, and D. Panario. "Locating modifications in signed data for partial data integrity". In: *Information Processing Letters* 115.10 (2015), pp. 731–737. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2015.02.014.

[Ida15] T. B. Idalino. "Using combinatorial group testing to solve integrity issues". MA thesis. Universidade Federal de Santa Catarina, Brazil, 2015. URL: https://repositorio.ufsc.br/handle/123456789/169646?show=full (visited on 08/13/2019).

[IM18] T. B. Idalino and L. Moura. "Efficient Unbounded Fault-Tolerant Aggregate Signatures Using Nested Cover-Free Families". In: *Combinatorial Algorithms*. Ed. by C. Iliopoulos, H. W. Leong, and W.-K. Sung. Lecture Notes in Computer Science 10979. Springer International Publishing, 2018, pp. 52–64. ISBN: 978-3-319-94667-2. DOI: 10.1007/978-3-319-94667-2_5.

[Int] Intel Corporation. *Intel Core i5-2430M Processor Specification*. URL: https://ark.intel.com/products/53450/Intel-Core-i5-2430M-Processor-3M-Cache-up-to-3_00-GHz (visited on 08/13/2019).

[Int12] Intel Corporation. *2nd Generation Intel Core Mobile Processor Datasheet, Volume 1*. 2012. URL: https://www-ssl.intel.com/content/www/us/en/processors/core/2nd-gen-core-family-mobile-vol-1-datasheet.html (visited on 05/29/2017).

[IR01] G. Itkis and L. Reyzin. "Forward-Secure Signatures with Optimal Signing and Verifying". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by J. Kilian. Lecture Notes in Computer Science 2139. Springer Berlin Heidelberg, 2001, pp. 332–354. ISBN: 978-3-540-42456-7. DOI: 10.1007/3-540-44647-8_20.

[Kai20] B. Kaidel. "Fault-Tolerance and Deaggregation Security of Aggregate Signatures". PhD thesis. Karlsruhe Institute of Technology (KIT), 2020. DOI: 10.5445/IR/1000105565.

[Kaw+05] N. Kawaguchi, N. Obata, S. Ueda, Y. Azuma, H. Shigeno, and K.-I. Okada. "Efficient log authentication for forensic computing". In: *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*. 2005, pp. 215–223. DOI: 10.1109/IAW.2005.1495955.

[KB79] R. Kannan and A. Bachem. "Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix". In: *SIAM Journal on Computing* 8.4 (1979), pp. 499–507. DOI: 10.1137/0208040.

[Ker83] A. Kerckhoffs. "La Cryptographie Militaire". In: *Journal des Sciences Militaires* 9 (1883), pp. 5–38. URL: https://archive.org/details/117Kerckhoffs (visited on 08/13/2019).

[Kil+05] E. Kiltz, A. Mityagin, S. Panjwani, and B. Raghavan. "Append-Only Signatures". In: *Automata, Languages and Programming*. Ed. by L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung. Lecture Notes in Computer Science 3580. Springer Berlin Heidelberg, 2005, pp. 434–445. ISBN: 978-3-540-31691-6. DOI: 10.1007/11523468_36.

[KL07] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. 1st ed. Chapman & Hall/CRC, 2007. ISBN: 1584885513.

[Kra00] H. Krawczyk. "Simple forward-secure signatures from any signature scheme". In: *CCS '00: Proceedings of the 7th ACM Conference on Computer and Communications Security*. Ed. by D. Gritzalis, S. Jajodia, and P. Samarati. Association for Computing Machinery, 2000, pp. 108–115. ISBN: 1-58113-203-4. DOI: 10.1145/352600.352617.

[KRS99] R. Kumar, S. Rajagopalan, and A. Sahai. "Coding Constructions for Blacklisting Problems without Computational Assumptions". In: *Advances in Cryptology — CRYPTO '99*. Ed. by M. J. Wiener. Lecture Notes in Computer Science 1666. Springer, 1999, pp. 609–623. ISBN: 3-540-66347-9. DOI: 10.1007/3-540-48405-1_38.

[KS06] K. Kent and M. Souppaya. *Guide to Computer Security Log Management*. NIST Special Publication 800-92. US National Institute of Science and Technology, 2006. DOI: 10.6028/NIST.SP.800-92.

[KS64] W. H. Kautz and R. C. Singleton. "Nonrandom binary superimposed codes". In: *IEEE Transactions on Information Theory* 10.4 (1964), pp. 363–377. DOI: 10.1109/TIT.1964.1053689.

[Lat85] D. C. Latham, ed. *Department of Defense Trusted Computer System Evaluation Criteria*. US Department of Defense, 1985. URL: http://csrc.nist.gov/publications/history/dod85.pdf (visited on 08/13/2019).

[Lin17] A. Lindqvist. "Privacy Preserving Audit Proofs". MA thesis. KTH Royal Institute Of Technology, Sweden, 2017.

[Lyn] B. Lynn. *The Pairing-Based Crypto Library*. URL: https://crypto.stanford.edu/pbc/ (visited on 08/13/2019).

[Lys+04] A. Lysyanskaya, S. Micali, L. Reyzin, and H. Shacham. "Sequential Aggregate Signatures from Trapdoor Permutations". In: *Advances in Cryptology — EUROCRYPT 2004*. Ed. by C. Cachin and J. L. Camenisch. Springer Berlin Heidelberg, 2004, pp. 74–90. ISBN: 978-3-540-24676-3. DOI: 10.1007/978-3-540-24676-3_5.

[Ma08] D. Ma. "Practical Forward Secure Sequential Aggregate Signatures". In: *ASIACCS '08: Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. Association for Computing Machinery, 2008, pp. 341–352. ISBN: 978-1-59593-979-1. DOI: 10.1145/1368310.1368361.

[Mer88] R. C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Ed. by C. Pomerance. Lecture Notes in Computer Science 293. Springer Berlin Heidelberg, 1988, pp. 369–378. ISBN: 978-3-540-48184-3. DOI: 10.1007/3-540-48184-2_32.

[MMM02] T. Malkin, D. Micciancio, and S. Miner. "Efficient Generic Forward-Secure Signatures with an Unbounded Number of Time Periods". In: *Advances in Cryptology — EUROCRYPT 2002*. Ed. by L. R. Knudsen. Lecture Notes in Computer Science 2332. Springer Berlin Heidelberg, 2002, pp. 400–417. ISBN: 978-3-540-43553-2. DOI: 10.1007/3-540-46035-7_27.

[MP13] G. A. Marson and B. Poettering. "Practical Secure Logging: Seekable Sequential Key Generators". In: *Computer Security – ESORICS 2013*. Ed. by J. Crampton, S. Jajodia, and K. Mayes. Lecture Notes in Computer Science 8134. Springer Berlin Heidelberg, 2013, pp. 111–128. ISBN: 978-3-642-40202-9. DOI: 10.1007/978-3-642-40203-6_7.

[MT07a] D. Ma and G. Tsudik. "Extended Abstract: Forward-Secure Sequential Aggregate Authentication". In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, 2007, pp. 86–91. ISBN: 0-7695-2848-1. DOI: 10.1109/SP.2007.18.

[MT07b] D. Ma and G. Tsudik. *Forward-Secure Sequential Aggregate Authentication*. Cryptology ePrint Archive, Report 2007/052. 2007. URL: https://eprint.iacr.org/2007/052.

[MT08] D. Ma and G. Tsudik. "A New Approach to Secure Logging". In: *Data and Applications Security XXII*. Ed. by V. Atluri. Lecture Notes in Computer Science 5094. Springer Berlin Heidelberg, 2008, pp. 48–63. ISBN: 978-3-540-70566-6. DOI: 10.1007/978-3-540-70567-3_4.

[MT09] D. Ma and G. Tsudik. "A New Approach to Secure Logging". In: *ACM Transactions on Storage (TOS)* 5.1 (2009), 2:1–2:21. ISSN: 1553-3077. DOI: 10.1145/1502777.1502779.

[MW01] D. Micciancio and B. Warinschi. "A Linear Space Algorithm for Computing the Hermite Normal Form". In: *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*. ISSAC '01. Association for Computing Machinery, 2001, pp. 231–236. ISBN: 1-58113-417-7. DOI: 10.1145/384101.384133.

[NDP17] M. Nieles, K. Dempsey, and V. Y. Pilliteri. *An Introduction to Information Security*. Revision 1. NIST Special Publication 800-12. US National Institute of Science and Technology, 2017. DOI: 10.6028/NIST.SP.800-12r1.

[PD18] T. Pulls and R. Dahlberg. "Steady - A Simple End-to-End Secure Logging System". In: *Secure IT Systems*. Ed. by N. Gruschka. Springer International Publishing, 2018, pp. 88–103. ISBN: 978-3-030-03638-6. DOI: 10.1007/978-3-030-03638-6_6.

[RFC5848] J. Kelsey, J. Callas, and A. Clemm. *Signed Syslog Messages*. RFC 5848. 2010. URL: https://tools.ietf.org/html/rfc5848 (visited on 01/30/2020).

[RFC6962] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962. 2013. URL: https://tools.ietf.org/html/rfc6962 (visited on 01/30/2020).

[RFC8446] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. 2018. URL: https://tools.ietf.org/html/rfc8446 (visited on 01/30/2020).

[Sch90] C.-P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89*. Ed. by G. Brassard. Lecture Notes in Computer Science 435. Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-97317-3. DOI: 10.1007/0-387-34805-0_22.

[Sch91] C.-P. Schnorr. "Efficient signature generation by smart cards". In: *Journal of Cryptology* 4.3 (1991), pp. 161–174. ISSN: 0933-2790. DOI: 10.1007/BF00196725.

[Sha85] A. Shamir. "Identity-Based Cryptosystems and Signature Schemes". In: *Advances in Cryptology — CRYPTO 1984*. Ed. by G. R. Blakley and D. Chaum. Lecture Notes in Computer Science 196. Springer Berlin Heidelberg, 1985, pp. 47–53. ISBN: 978-3-540-15658-1. DOI: 10.1007/3-540-39568-7_5.

[Sho] V. Shoup. *NTL: A Library for doing Number Theory*. URL: http://shoup.net/ntl/ (visited on 08/13/2019).

[SK98] B. Schneier and J. Kelsey. "Cryptographic Support for Secure Logs on Untrusted Machines". In: *Proceedings of the 7th USENIX Security Symposium*. Ed. by A. D. Rubin. USENIX Association, 1998. URL: https://www.usenix.org/conference/7th-usenix-security-symposium/cryptographic-support-secure-logs-untrusted-machines (visited on 08/13/2019).

[SK99] B. Schneier and J. Kelsey. “Secure Audit Logs to Support Computer Forensics”. In: *ACM Transactions on Information and System Security* 2.2 (1999), pp. 159–176. ISSN: 1094-9224. DOI: 10.1145/317087.317089. URL: https://www.schneier.com/paper-auditlogs.pdf (visited on 08/13/2019).

[SMD14] A. Saxena, J. Misra, and A. Dhar. “Increasing Anonymity in Bitcoin”. In: *Financial Cryptography and Data Security*. Ed. by R. Böhme, M. Brenner, T. Moore, and M. Smith. Lecture Notes in Computer Science 8438. Springer, 2014, pp. 122–139. ISBN: 978-3-662-44773-4. DOI: 10.1007/978-3-662-44774-1_9.

[Son01] D. X. Song. “Practical Forward Secure Group Signature Schemes”. In: *CCS ’01: Proceedings of the 8th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2001, pp. 225–234. ISBN: 1-58113-385-5. DOI: 10.1145/501983.502015.

[Ste] W. Stein. *SageMath*. URL: https://www.sagemath.org/ (visited on 08/13/2019).

[Wat+04] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. “Building an Encrypted and Searchable Audit Log”. In: *The 11th Annual Network and Distributed System Security Symposium*. The Internet Society, 2004. ISBN: 1-891562-18-5. URL: https://www.ndss-symposium.org/ndss2004/building-encrypted-and-searchable-audit-log/.

[Wei] S. H. Weintraub. *Algebra: An Approach via Module Theory—Errata*. URL: https://www.lehigh.edu/~shw2/algebra-errata.pdf (visited on 08/13/2019).

[XCO05] W. Xu, D. W. Chadwick, and S. Otenko. “A PKI based secure audit web server”. In: *IASTED Communications, Network and Information and CNIS*. 2005.

[YP09] A. A. Yavuz and N. Peng. “BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems”. In: *Annual Computer Security Applications Conference — ACSAC 2009*. 2009, pp. 219–228. ISBN: 978-1-4244-5327-6. DOI: 10.1109/ACSAC.2009.28.

[YPR12a] A. A. Yavuz, N. Peng, and M. K. Reiter. “BAF and FI-BAF: Efficient and Publicly Verifiable Cryptographic Schemes for Secure Logging in Resource-Constrained Systems”. In: *ACM Transactions on Information and System Security* 15.2 (2012), 9:1–9:28. ISSN: 1094-9224. DOI: 10.1145/2240276.2240280.

[YPR12b] A. A. Yavuz, N. Peng, and M. K. Reiter. “Efficient, Compromise Resilient and Append-Only Cryptographic Schemes for Secure Audit Logging”. In: *Financial Cryptography and Data Security*. Ed. by A. D. Keromytis. Lecture Notes in Computer Science 7397. Springer Berlin Heidelberg, 2012,

pp. 148–163. ISBN: 978-3-642-32945-6. DOI: 10.1007/978-3-642-32946-3_12.

[YR11] A. A. Yavuz and M. K. Reiter. *Efficient, Compromise Resilient and Append-Only Cryptographic Schemes for Secure Audit Logging*. Tech. rep. TR-2011-21. North Carolina State University. Department of Computer Science, 2011. URL: http://www.lib.ncsu.edu/resolver/1840.4/4284 (visited on 01/29/2020).

[ZWW03] J. Zhang, Q. Wu, and Y. Wang. "A Novel Efficient Group Signature Scheme with Forward Security". In: *Information and Communications Security*. Ed. by S. Qing, D. Gollmann, and J. Zhou. Lecture Notes in Computer Science 2836. Springer Berlin Heidelberg, 2003, pp. 292–300. ISBN: 978-3-540-20150-2. DOI: 10.1007/978-3-540-39927-8_27.

Appendices

A. Example CFF Instantiation

We briefly present a simple instantiation (due to [KRS99, Section 5]) of a cover-free family.

Let $k \in \mathbb{N}_0$, p be a prime number, and $\mathbb{F}_p$ be the finite field of integers modulo p. The CFF's base set $\mathcal{S}$ is the set of all points in the plane over $\mathbb{F}_p$, i.e.

$$\mathcal{S} = \mathbb{F}_p^2.$$

We construct the blocks B of the CFF as follows. Fix a polynomial $f \in \mathbb{F}_p[X]$ of degree at most k. The block B associated with f is the set of all points on the graph of f, i.e.

$$B_f = \{(x, f(x)) \colon x \in \mathbb{F}_p\}.$$

The set of all blocks is obtained by repeating this process for all polynomials f of degree at most k, i.e.

$$\mathcal{B} = \{B_f \colon f \in \mathbb{F}_p[X] \text{ and } \deg(f) \leq k\},$$

where $\deg(f)$ is the degree of f.

Since the graphs of two distinct polynomials $f, g \in \mathbb{F}_p[X]$ can intersect on at most k points, we have that $|B_f \setminus B_g| \geq p - k$. This is illustrated in Figure A.1. Extending this argument to up to d polynomials $g_1, \ldots, g_d \in \mathbb{F}_p[X]$ (with $f \neq g_i$ for all $i \in [d]$), we have

$$\left| B_f \setminus \left(\bigcup_{i=1}^{d} B_{g_i} \right) \right| \geq p - dk.$$

Hence, if $p > dk$, then the union of the d sets B_{g_i} does not cover B_f, and thus $\mathcal{F} = (\mathcal{S}, \mathcal{B})$ is a d-CFF. This CFF has $|\mathcal{S}| = p^2$ and $|\mathcal{B}| = p^{k+1}$. It is easy to define an order on $\mathcal{B}$ by interpreting the coefficients of a polynomial f as digits (in base p) of a natural number, and then ordering the functions by the standard order on $\mathbb{N}$. The set $\mathcal{S} = \mathbb{F}_p^2$ can be ordered analogously. Thus, we may consider this CFF to be ordered.

Table A.1 shows some parameters that might be used to instantiate the CFF described above. For our construction given in Chapter 5, the number of blocks $|\mathcal{B}|$ corresponds to the number of log entries that can be handled by the scheme, the number r corresponds to the number of (sequentially aggregate) signatures to be kept, and d corresponds to the maximum number of log entries that can be modified by an attacker without affecting the verifiability of unmodified log entries.

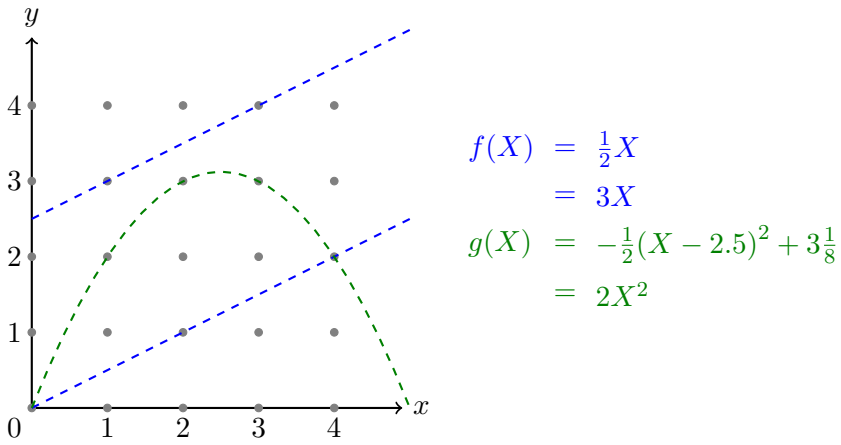

Figure A.1.: Illustration of two blocks of the CFF construction with $p = 5$ and $k = 2$. The blue line illustrates the graph of the polynomial $f(X) = \frac{1}{2}X = 3X \in \mathbb{F}_p[X]$, while the green line depicts the graph of the polynomial $g(X) = -\frac{1}{2}(X-2.5)^2+3\frac{1}{8} = 2X^2 \in \mathbb{F}_p[X]$. The two polynomials intersect in the points $(0,0)$ and $(4,2)$, hence $|B_f \setminus B_g| = 3 \geq p-k = 3$. While $\mathbb{F}_p$ is not a continuous but a discrete set, we have nonetheless drawn (dashed) lines to depict the graphs in order to appeal to the reader's intution.

Table A.1.: Example parameters for the CFF construction of [KRS99]. Taken from [Har+16] and extended by additional rows.

p	k	d	$r = \lvert\mathcal{S}\rvert$	$n = \lvert\mathcal{B}\rvert$
5	2	2	25	125
11	2	5	121	1331
17	2	8	289	4913
17	4	4	289	$\approx 1.42 \cdot 10^6$
29	2	14	841	24389
53	2	26	2809	148877
101	2	50	10201	$\approx 1.03 \cdot 10^6$
251	3	83	63001	$\approx 3.97 \cdot 10^9$
401	4	100	160801	$\approx 9.43 \cdot 2^{40}$
503	5	100	253009	$\approx 14.4 \cdot 2^{50}$
601	6	100	361201	$\approx 1.54 \cdot 2^{64}$
1021	2	510	1042441	$\approx 1.06 \cdot 10^9$
1213	24	50	1471369	$\approx 1.08 \cdot 2^{256}$
2243	24	100	5031049	$\approx 1.01 \cdot 2^{256}$

B. Proof of Robustness

This appendix shows that the logging scheme from Section 5.4.2 (see page 112 and following) is *robust*, as specified in Definition 5.18 (see page 110). We begin with a straightforward lemma, which almost immediately implies the fault-tolerance of our scheme.

In the following, let LS be the logging scheme with list verification defined in Section 5.4.2 and AS, FS be the underlying signature schemes, respectively.

Lemma B.1. Let $C = ((\mathsf{pk}, t_i, m_i))_{i=1}^{l}$ be a claim sequence (for $l \in \mathbb{N}_0$) and $\tau = (\sigma, s)$ be regular for C, and $\mathsf{pk} = (\mathsf{pk}_{\mathsf{AS}}, \mathsf{pk}_{\mathsf{FS}})$ be the public key in C. Then σ is regular for $C_{\mathsf{AS}} = ((\mathsf{pk}_{\mathsf{AS}}, t_i, (i, m_i)))_{i=1}^{l}$, and s is regular for $\mathrm{len}(C)$.

Proof. We show the claim by induction. For the induction start, let $l = 0$. Then $C = ()$, and $\tau = (\sigma_0 = \lambda, s_0 = \mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}, 0))$, where $\mathsf{sk}_{\mathsf{FS}}$ is the secret key for FS (for the first epoch) generated by KeyGen. Thus by definition, σ_0 is regular for $C_{\mathsf{AS}} = ()$, and s_0 is regular for $\mathrm{len}(C) = 0$.

Now assume that the lemma's claim holds for all claim sequences of a specific length $l \in \mathbb{N}_0$. We show the claim holds for all claim sequences of length $l + 1$, too.

Let C be an arbitrary claim of length $l+1$, C' be the length-l prefix of C, $c_{l+1} = C[l+1] = (\mathsf{pk}, t_{l+1}, m_{l+1})$ be the last claim in C, and $\tau = (\sigma, s)$ be a regular signature for C. Since τ is regular, (C, τ) is in the output of the process described in Definition 5.15. Hence C is either the result of a modification done by Update in step 3 of the process, or the result of concatenating some claim to C in step 2c. In either case, C is the result of the concatenation of C' and c_{l+1}, and τ is computed as $\mathsf{Append}(\mathsf{sk}_{t_{l+1}}, C', \tau', m_{l+1})$, where $\tau' = (\sigma', s')$ is regular for C'.

By the induction hypothesis, σ' is regular for $C'_{\mathsf{AS}} = ((\mathsf{pk}_{\mathsf{AS}}, t_i, (i, m_i)))_{i=1}^{l}$. Thus, since Append computes σ as $\mathsf{AS.AggSign}(\mathsf{sk}_{\mathsf{AS}}, C'_{\mathsf{AS}}, \sigma', (l+1, m_{l+1}))$, we have that σ is regular for $C_{\mathsf{AS}} = C'_{\mathsf{AS}} \,\|\, (\mathsf{pk}_{\mathsf{AS}}, t_{l+1}, (l+1, m_{l+1}))$, as claimed.

Moreover, Append computes s as $\mathsf{FS.Sign}(\mathsf{sk}_{\mathsf{FS}}, l+1)$, so, again, s is regular for $l + 1 = \mathrm{len}(C)$ by definition. This concludes the proof. □

Using the lemma above, we now show the fault tolerance of our scheme.

Corollary B.2 (Fault Tolerance)**.** LS is d-fault-tolerant if AS is d-fault-tolerant.

Proof. Let C be a claim sequence, τ be a regular signature for C, and C' be a claim sequence containing d errors with respect to C. Let C_{AS} be the claim sequence built by the $\mathsf{ValidEntries}$ algorithm when called with the arguments C and τ, and C'_{AS} be the claim sequence built by $\mathsf{ValidEntries}$ when called with the parameters C' and τ. If C' contains d errors with regard to C, then C'_{AS} contains d errors with respect to C_{AS},

too, and these positions are the same as for C' and C. Since AS is d-fault-tolerant and σ is regular for C_{AS} (see Lemma B.1), $\mathsf{AS.Verify}(C'_{\mathsf{AS}}, \sigma)$ returns 1 on at least all positions i where $C_{\mathsf{AS}}[i] = C'_{\mathsf{AS}}[i]$, and hence where $C[i] = C'[i]$. This output also is the return value of $\mathsf{ValidEntries}$, and thus LS is d-fault-tolerant, too. □

We have thus shown that our logging scheme is fault-tolerant if it is instantiated with a fault-tolerant key-evolving SAS scheme. Towards showing the correctness and robustness of our scheme, we still need to show that it is verification-consistent. This proof is straightforward, too:

Lemma B.3 (Verification Consistency)**.** LS is verification-consistent.

Proof. Let C be an arbitrary claim sequence and τ be a signature. $\mathsf{VerifyLog}$ only outputs 1 iff $\mathsf{ValidEntries}(C, \tau) = 1^{\mathrm{len}(C)}$. Hence, we have

$$\mathsf{ValidEntries}(C, \tau) \neq 1^{\mathrm{len}(C)} \implies \mathsf{VerifyLog}(C, \tau) \neq 1.$$

This is logically equivalent to its converse

$$\mathsf{VerifyLog}(C, \tau) = 1 \implies \mathsf{ValidEntries}(C, \tau) = 1^{\mathrm{len}(C)}. \qquad \square$$

We may now prove our theorem from Section 5.4.2:

Theorem 5.20 (Correctness and Robustness)**.** If FS is correct and AS is correct, then LS is correct. If moreover AS is fault-tolerant for a $d > 0$, then LS is d-fault-tolerant and hence robust.

Proof. We begin by showing correctness. We have already shown that LS is verification-consistent, which is the first requirement for correctness. For the second requirement, we need to show that for all claim sequences C and signatures $\tau = (\sigma, s)$ such that τ is regular for C, we have that $\mathsf{VerifyLog}(C, \tau) = 1$.

For the first check performed by $\mathsf{VerifyLog}$, observe that the process for creating regular signatures (see Definition 5.15) only outputs claim sequences where all public keys are the same. Thus, the first check passes for all regularly created claim sequences C.

The second and third check of $\mathsf{LS.VerifyLog}$ passes by construction of the Update algorithm of LS, which adds the respective epoch markers and thus makes sure that $t_{i+1} - t_i \in \{0, 1\}$.

By Lemma B.1, s is regular for $\mathrm{len}(C)$. Since FS is correct, the next check in $\mathsf{LS.VerifyLog}$ will pass, too. Finally, by Lemma B.1, σ is regular for the corresponding sequence C_{AS} built by $\mathsf{ValidEntries}$. Since AS is correct $\mathsf{AS.Verify}$ outputs $1^{\mathrm{len}(C)}$, and thus $\mathsf{LS.ValidEntries}$ does, too.

Thus, the final check of $\mathsf{LS.VerifyLog}$ passes and the algorithm outputs 1. We have thus shown the correctness of LS. All that remains to show is the robustness of LS.

Assume that AS is d-fault-tolerant for a $d > 0$ (and FS is correct). Then, by Corollary B.2, LS is d-fault-tolerant, too, and hence it is robust, as claimed. □

C. List of the Author's Publications

The author has participated in the research underlying the following formal, peer-reviewed publications:

[Bro+17] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. "Concurrently Composable Security with Shielded Super-Polynomial Simulators". In: *Advances in Cryptology – EUROCRYPT 2017*. Ed. by J.-S. Coron and J. B. Nielsen. Springer International Publishing, 2017, pp. 351–381. ISBN: 978-3-319-56620-7. DOI: 10.1007/978-3-319-56620-7_13.

[Har+16] G. Hartung, B. Kaidel, A. Koch, J. Koch, and A. Rupp. "Fault-Tolerant Aggregate Signatures". In: *Public-Key Cryptography — PKC 2016*. Ed. by C. Cheng, K. Chung, G. Persiano, and B. Yang. Lecture Notes in Computer Science 9614. Springer, 2016, pp. 331–356. ISBN: 978-3-662-49383-0. DOI: 10.1007/978-3-662-49384-7_13.

[Har+17a] G. Hartung, M. Hoffmann, M. Nagel, and A. Rupp. "BBA+: Improving the Security and Applicability of Privacy-Preserving Point Collection". In: *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2017, pp. 1925–1942. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134071.

[Har+17b] G. Hartung, B. Kaidel, A. Koch, J. Koch, and D. Hartmann. "Practical and Robust Secure Logging from Fault-Tolerant Sequential Aggregate Signatures". In: *Provable Security*. Ed. by T. Okamoto, Y. Yu, M. H. Au, and Y. Li. Lecture Notes in Computer Science 10592. Springer International Publishing, 2017, pp. 87–106. ISBN: 978-3-319-68637-0. DOI: 10.1007/978-3-319-68637-0_6.

[Har16a] G. Hartung. "Secure Audit Logs with Verifiable Excerpts". In: *Topics in Cryptology — CT-RSA 2016*. Ed. by K. Sako. Lecture Notes in Computer Science 9610. Springer, 2016, pp. 183–199. ISBN: 978-3-319-29484-1. DOI: 10.1007/978-3-319-29485-8_11.

[Har17] G. Hartung. "Attacks on Secure Logging Schemes". In: *Financial Cryptography and Data Security*. Ed. by A. Kiayias. Springer International Publishing, 2017, pp. 268–284. ISBN: 978-3-319-70972-7. DOI: 10.1007/978-3-319-70972-7_14.

In addition, the following informal publications include contributions by the author:

[Bro+16] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. *Concurrently Composable Security With Shielded Super-polynomial Simulators*. Cryptology ePrint Archive, Report 2016/1043. https://eprint.iacr.org/2016/1043. 2016.

[Har16b] G. Hartung. *Secure Audit Logs with Verifiable Excerpts – Full Version*. Cryptology ePrint Archive, Report 2016/283. https://eprint.iacr.org/2016/283. 2016.

Copyright

As indicated in Section 1.4, this thesis reproduces significant amounts of content and text copied verbatim (or with only slight modifications) from [Har16a; Har+17b; Har17; Har16b].

The content and text is reproduced in this thesis in accordance with the copyright agreements between the author and Springer International Publishing (for [Har16a; Har+17b]) and the International Financial Cryptography Association (for [Har17]). The publication [Har16b] (as the full version of [Har16a]) is also affected by the copyright agreement between the author and Springer International Publishing, and parts of this publication are reproduced here in accordance with the copyright agreement, too.